Versatile Event Correlation with Algebraic Effects

OLIVER BRAČEVAC, TU Darmstadt, Germany
NADA AMIN, University of Cambridge, UK
GUIDO SALVANESCHI, TU Darmstadt, Germany
SEBASTIAN ERDWEG, Delft University of Technology, The Netherlands
PATRICK EUGSTER, University of Lugano, Switzerland; TU Darmstadt, Germany; Purdue University, USA
MIRA MEZINI, TU Darmstadt, Germany

We present the first language design to uniformly express variants of n-way joins over asynchronous event streams from different domains, e.g., stream-relational algebra, event processing, reactive and concurrent programming. We model asynchronous reactive programs and joins in direct style, on top of algebraic effects and handlers. Effect handlers act as modular interpreters of event notifications, enabling fine-grained control abstractions and customizable event matching. Join variants can be considered as cartesian product computations with "degenerate" control flow, such that unnecessary tuples are not materialized a priori. Based on this computational interpretation, we decompose joins into a generic, naïve enumeration procedure of the cartesian product, plus variant-specific extensions, represented in terms of user-supplied effect handlers. Our microbenchmarks validate that this extensible design avoids needless materialization. Alongside a formal semantics for joining and prototypes in Koka and multicore OCaml, we contribute a systematic comparison of the covered domains and features.

CCS Concepts: • **Software and its engineering → Control structures**; **Patterns**; **Coroutines**; **Semantics**;

Additional Key Words and Phrases: event correlation, complex event processing, joins, asynchrony, algebraic effect handlers, Koka, multicore OCaml

ACM Reference Format:
Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile Event Correlation with Algebraic Effects. *Proc. ACM Program. Lang.* 2, ICFP, Article 67 (September 2018), 31 pages. https://doi.org/10.1145/3236762

1 INTRODUCTION

Events notify a software system of incidents in its dynamic environment. Examples of event sources are sensors, input devices, or network hosts. *Event correlation* means to make deductions about the state of the environment, given observations of events from different sources over time. For example, by correlating batches or streams of events, computer systems drive cars, trade stocks, or give recommendations; by correlating input events smartphones and tablet devices interpret touch gestures. In the words of Luckham [2001]: "*Events are related in various ways, by cause, by timing, and by membership*". Since events are asynchronous, computing such event relations amounts to defining n-way *synchronizers* (or *joins*) over events. To avoid teasing out subtle distinctions, we use the terms *join* and *correlation* interchangeably.

Different communities have thus far invented various specialized abstractions for joins. On the systems side, there are *complex event processing (CEP)* and *stream processing* systems [Cugola and Margara 2012]. On the programming languages side, there are *reactive programming* [Bainomugisha

2475-1421/2018/9-ART67
https://doi.org/10.1145/3236762

et al. 2013] and *concurrent programming languages*, e.g., [Benton et al. 2004; Conchon and Le Fessant 1999; Fluet et al. 2008; Fournet and Gonthier 1996; Reppy 1991]. Common to all families is that they support some sort of high-level specifications of joins, e.g., as declarative patterns or queries.

However, a commonly agreed-upon semantics for joins (and by extension event correlation) remains elusive. Not only is the number of features for relating events staggering (cf. the surveys by Bainomugisha et al. [2013]; Cugola and Margara [2012]), but already a single feature can be interpreted quite differently between systems and even within one family (e.g., time windows [Dindar et al. 2013]). This lack of clarity is an obstacle for choosing the right language/system: It is hard to determine its adequacy for expressing a desired event correlation behavior. Moreover, since each system provides specialized abstractions that cannot be easily changed, some applications may not find any adequate language/system to meet their requirements.

Motivated by these observations, the goal of our work is to contribute a language design and structured programming abstractions for defining n-way event joins with customizable, extensible, and composable semantics, which we call *versatile joins*. More specifically, we aim to enable:

- **mix-and-match style compositions** of features for relating events (from across domains), customization in the sense that **features can be reinterpreted** (overloaded) to express subtle differences, and **extensibility with new features**.
- **controllable matching behavior:** To model versatile joins, one needs to consider all possible ways joins can pattern match and process n infinite push streams. There are plenty ways of doing this: Aligning, skipping, duplicating, timing, dependent on event values, or any combination thereof. To manage this complexity, the language needs generic control abstractions for coordination and alignment of streams.
- **direct style specifications:** Asynchrony of events encourages programming in some form of continuation-passing style (CPS), due to inversion of control. This is error-prone and does not scale in the large, leading to "callback-hell" [Edwards 2009]. Hence, the language design should enable users to express asynchronous computations in direct style. This is easier to understand and more natural than working with continuations directly. Even more so in the case of n-way *joins*, where n interdependent continuations need to be coordinated.

This paper proposes Cartesius – a domain-specific language for programming versatile joins over infinite event sequences that satisfies the above goals. Its design is guided by an informal intuition of *what a join is*:

Proposition 1.1 (Intuitive Interpretation). *A join (correlation) is a restriction of the cartesian product over the inputs.*

This view of joins is deliberately open-ended, since we aim for extensibility and customizability. We consider the pure cartesian product as the most general correlation, confining the space of possible results. Any other variant of event correlation is a restriction of the cartesian product, where a "restriction" can take many forms, e.g., a simple attribute filter on values, or a nondeterministic selection of values, which is an *effect*. We support user-defined effects as a way to specify, customize, and add new restrictions.

Proposition 1.1 defines what joins "are" in a way that is not far from a denotational semantics, e.g., relational algebra. At the same time, this informal view is already useful to derive a way of *doing joins*, which is simple yet realistic for implementation in a programming language:

Proposition 1.2 (Computational Interpretation). *A join computation is an enumeration of the cartesian product over the inputs, with (user-defined) side effects influencing how the computation proceeds.*

Following this computational interpretation, the core of Cartesius provides a generic cartesian product implementation. Yet, despite working with such a naïve and expensive generic component, effects enable us to obtain computationally efficient variants of correlations in Cartesius. That is, the cartesian product retains all observed events forever and materializes all combinations of events. In practice, though, only a small fraction of event combinations is relevant to the correlation computation, e.g., when zipping streams. User-defined effects can manipulate the control flow so that irrelevant event combinations are *never* materialized.

For language-level support of effects, Cartesius employs algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009]. They adequately address our requirements on the language design. First, *algebras* of effectful operations are well-suited for supporting extensibility and customizability: They enable purely functional definitions of effects, compose freely, and enable language extensions/customizations as libraries, *without changes to compiler or runtime.* Second, *effect handlers* capture control flow akin to coroutines [de Moura and Ierusalimschy 2009], which lets users express asynchronous computations in direct style [Leijen 2017a], diminishing the pains of inversion of control and supporting custom control abstractions.

In the remainder of the paper, we make the following contributions:

- a high-level example-driven overview of Cartesius in Section 2;
- a formal semantics of a core calculus for effect handlers with parametric and effect polymorphism, λ_{cart}, in Section 3;
- a library-based design of Cartesius, as an embedding into λ_{cart}. The design features novel uses of algebraic effects and handlers: we encode event notifications, reactive programs, implicit variables and control abstractions for fine-grained coordination of asynchronous computations. We implemented the design in both Koka and multicore OCaml (Section 4).
- an evaluation of Cartesius's expressivity, through a systematic comparison with works surveyed across the CEP/streaming engines, reactive and concurrent programming languages in Section 5;
- a small-scale, preliminary study to quantify and gain insights on the effectiveness of the computational interpretation in Section 6;
- a discussion on experiences gained from implementing Cartesius in real algebraic effects languages and potential future research directions in Section 7.

To the best of our knowledge, this is the first work to (a) enable programmable event correlation with clearly defined semantics – a long standing problem for event languages – and (b) to identify and adopt algebraic effects and handlers as a viable method for solving the problem.

2 AN OVERVIEW OF CARTESIUS

This section presents a high-level overview of Cartesius from the perspective of end users who need to specify and customize joins. We assume a host language akin to ML with algebraic effects. Furthermore, we outline how our computational interpretation of joins integrates event correlation features from different domains.

2.1 Event Sources

There are two kinds of asynchronous event sources in Cartesius computations: Input event sources that connect to the external world, e.g., sensors, input devices, etc., and output event sources that are defined by correlation patterns. Both are called *reactives.*

Cartesius embodies reactives as values of the parametric data type $\mathsf{R}[T]$, where T is the type of the *payload* carried by events. An event of type T in Cartesius is actually a product type $(T \times \mathsf{Time})$ for some time representation Time. That is, an event embodies the evidence of something that *happened* at a particular point in time. This is unlike works on reactive programming (e.g., [Cave et al. 2014]), which view events of type T as the *potential to yield* a value in terms of an eventually modality $\Diamond[T]$. These two views are not incompatible, though. In fact, both show up in the definition of R[T], which corresponds to an infinite sequence of events in the first sense:

$$\mathsf{R}[T] \approx \mu X.\Diamond[(T \times \mathsf{Time}) \times X].$$

That is, an event source of Cartesius *potentially yields events*, one after another. We adopt this representation from [Elliott 2009] and discuss its precise definition and benefits in Section 3.

Many CQL-like query languages [Arasu et al. 2004; Krämer and Seeger 2009] and CEP languages [Demers et al. 2006; Diao et al. 2007] also exhibit timing information on data streams and events. The design space for representing time in these languages ranges from singular time stamps to sets of time stamps per event, trading off accuracy and space requirements [White et al. 2007]. Cartesius employs intervals for constant space usage, while still enabling an approximation of an event's history.

2.2 Expressing Event Relations in Direct Style

In Cartesius, we express versatile event correlation through declarative *correlation patterns*, which correlates events from one to many reactives, thereby transforming event data, or filtering events. A basic example is the following pattern:

```
let mouse: R[Int×Int] = ... //mouse input
let even_mouse_product: R[Int] =
 correlate
  { p from mouse
    where (fst p) mod 2 = 0
    yield (fst p) * (snd p) }
```

where the syntactic form **correlate** delimits the pattern body between the curly braces. This correlation pattern defines a reactive *even_mouse_product* correlating mouse position events, which are integer pairs. We filter the positions with an even first component using **where** and multiply their components within **yield**. The overall result of the correlation is a reactive emitting all multiplications of mouse positions satisfying the condition.

Note that this example is an asynchronous computation, but its syntax maintains the intuitive direct style of comprehension syntax similar to stream query languages. That is, *even_mouse_product* looks demand-driven (or pull-based), but it is reacting to event observations from the external mouse input (push-based). Internally, inversion of control is present, but it remains hidden to end-users.

In general, event correlations are expressed as n-way correlation patterns of the form:

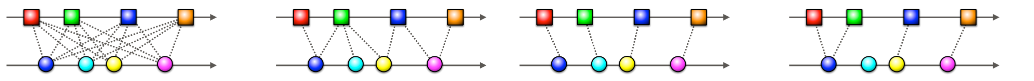
$$\textbf{correlate}\,\{\, x_1 \textbf{ from } r_1;\ldots;x_n \textbf{ from } r_n \textbf{ where } p \textbf{ yield } e \,\}$$

which intuitively represents a transformation[1]

$$\mathsf{R}[T_1] \Rightarrow \ldots \Rightarrow \mathsf{R}[T_n] \Rightarrow \mathsf{R}[T_{n+1}]$$

that forms an output reactive from n input reactives. The elements of the pattern syntax reflect this accordingly: (1) we bind individual event variables from input reactives with x_i **from** r_i, in the scope p and e, (2) intensionally specify relations among the events $(x_i)_{1\leq i\leq n}$ in the optional predicate **where** p, and then (3) apply a transformation **yield** e: $T_1 \times \ldots \times T_n \Rightarrow T_{n+1}$ to event combinations that are in the relation.

[1]We write $A \Rightarrow B$ to indicate transformations in a broad, informal sense, and $A \rightarrow B$ for proper function types.



(a) *cartesian* (b) *combineLatest* (c) *zip* (d) *affinely & combineLatest*

Fig. 1. Example correlations over two reactives, represented by arrows. Colored marbles represent events. Dashed lines indicate materialized pairings. The marble diagram notation is adopted from the Rx documentation.

2.3 Computational Interpretation

Here, we motivate the cartesian product with side effects view (Proposition 1.2). The most basic correlation pattern we can define in Cartesius is an n-way cartesian product, i.e., the *cartesian* definition in Figure 2, top left, which produces all combinations $\langle x,y \rangle$ from the *left* and *right* reactives (Figure 1a). Predicates as in the preceding example reduce the number of generated combinations, i.e., *restrict* the cartesian product. This is a simple and intuitive way to think about the semantics of event correlation and close to a relational algebra interpretation.

We also want to derive the operational behavior of event correlation as a realistic basis for concrete implementations in a programming language. The challenge is to keep the specification of the operational behavior simple and extensible, ideally, close to the intuitive relational view while achieving efficiency of the computation. A naïve implementation would generate all combinations and then test against the predicate, which is expensive and leads to space leaks [Krishnaswami 2013; Liu and Hudak 2007; Mitchell 2013]. Moreover, due to the asynchrony of reactives, we are forced to observe event notifications one-by-one. Hence, applying lazy techniques from demand-driven computation would not avoid the issue.

Our solution *a priori* avoids generating superfluous combinations. Specifically, we propose exposing overloadable, *user-defined effects* in the cartesian product computation. Simply by reinterpreting its effects, we force the computation into a specific operational behavior, so that the search space of combinations is cut down. In this way, end users can work with a generic, naïve generate-then-test implementation and turn it into a specialized, efficient computation.

For the concrete implementation of the operational behavior sketched above, Cartesius uses an effect system based on Plotkin et. al's algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009]. Some elements of the correlation pattern syntax desugar into effect operations. One such element is e.g., **yield**. By itself, **yield** has no semantics (i.e., implementation) – it requires a run time context (a handler) in which it is interpreted. For now, it is not necessary to understand this abstraction in detail. The only thing of note is that programmers can write and apply effect handlers that give implementations to effect operations as a form of dynamic overloading. Effect handlers are the main programming abstraction for customizing and specializing joins. We explain the basics of effects and handlers in Section 3 and how they are used for event correlation in Section 4.

2.4 Customizing Matching Behavior with Algebraic Effects

With the computational interpretation sketched above, modeling a specific correlation semantics becomes a matter of adding the right mixture of effect handlers. This aspect of the language design is crucial for defining extensions and mix-and-match style compositions of features. In the following, we exemplify this by specifying different correlation behaviors in Cartesius using the effect mechanism.

As a first example, we discuss the *combineLatest* combinator on asynchronous event sequences, e.g., as featured in the Reactive Extensions (Rx) library [ReactiveX [n. d.]]. As illustrated in Figure 1b, this combinator weakly aligns its inputs and always combines the *most recently* observed events.

```
1 let cartesian: R[A] → R[B] → R[A × B] = λleft. λright.
2   correlate
3     { x from left; y from right
4       yield ⟨x,y⟩ }
```

```
1 let combineLatest = λleft.λright.
2   let implicit ?restriction = (mostRecently left)
3     ⊞ (mostRecently right)
4   in cartesian left right
```

```
1 let zip = λleft. λright.
2   let implicit ?restriction = (mostRecently left)
3     ⊞ (mostRecently right)
4     ⊞ (aligning left right)
5   in cartesian left right
```

```
1 let affine_latest = λleft. λright.
2   let implicit ?restriction = (affinely left)
3     ⊞ (mostRecently left)
4     ⊞ (mostRecently right)
5   in cartesian left right
```

Fig. 2. Corresponding Cartesius code for Figure 1.

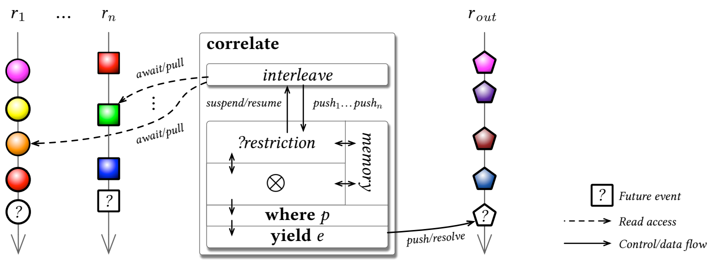


Fig. 3. Overview: Computational elements of correlation patterns.

Figure 2, top right, shows its corresponding Cartesius definition. In Line 2-3, we apply the combinator *mostRecently* to both *left* and *right* reactives. This combinator creates an effect handler that acts on its argument reactive (*left* and *right* in the example). The ⊞ operator composes the two effect handlers, so that both apply to the correlation. We may read this line as "inject this (compound) effect handler into the cartesian product computation". The handlers are injected through an *implicit variable ?restriction* into the computation. We use implicit variables starting with '*?*' for injecting dependencies (Section 3.1.4).

Correlation patterns always depend on the implicit variable *?restriction*, which by default is bound such that there is no restriction. During the computation of a correlation pattern, the effect injected by *?restriction* is applied. In the case of *combineLatest*, the injected effects force the cartesian product to provision only one element per input reactive, discarding events other than the latest one.

Pattern translation. Above, we specified a custom correlation behavior as a restriction of the cartesian product. To provide readers an overview of how this computation is structured underneath the surface, Figure 3 depicts the computational building blocks into which Cartesius translates n-way correlation patterns. Blocks designate computations, which interact by control or data transfer, indicated by solid arrows. A **correlate** pattern results in a number of sub-computations, which we describe top to bottom in the following.

The *interleave* sub-computation is a collection of n threads/strands, each independently iterating over one of the n input reactives. The dashed lines indicate that accessing the events of a reactive is potentially blocking. A reactive always has a finite prefix of materialized events and a tail that is yet-to-arrive in the future (indicated by question mark). For materialized input events, iteration proceeds in direct style, but suspends at the future tail with its continuation/callback, until the environment asynchronously materializes the tail. The i-th interleaved thread exposes the events to the join by performing a distinct $push_i$ effect to the building blocks downwards.

The user-supplied *?restriction* computation handles (i.e., observes) these $push_i$ effects. It embodies a custom synchronization logic for aligning the reactives, in the role of a coordinator or "puppeteer". For this purpose, *?restriction* has the power to suspend/resume individual strands. Another responsibility of *?restriction* is to interact with the fixed cartesian product computation ($\otimes$) – control shifts between the two as a form of coroutine [de Moura and Ierusalimschy 2009]. *?restriction* and $\otimes$ use a shared *memory* as communication medium to control the event combinations that $\otimes$ will generate. The default *?restriction* (unrestricted cartesian product) will store all events it observes from *interleave* in *memory* for further processing by $\otimes$. The *mostRecently ?restriction* introduced above would only store the last observed event in *memory* and truncate older events. Once $\otimes$ generates event combinations, these pass into the filtering and transformation steps as specified by **where** and **yield**. Finally, **yield**ed output events are materialized into an output reactive.

Composing custom correlation behaviors. To illustrate how combinators can be composed, consider the following correlation pattern that specifies the well-known *zip* combinator (Figure 1c). It exhibits a stricter alignment than *combineLatest* by adding another effect handler (*aligning left right*) in Figure 2, bottom left. The (*aligning*) handler changes the *selection behavior* of the cartesian product so that events from *left* and *right* are processed in lockstep. That is, if the correlation computation receives the next event from *left*, it will not process further *left* events until the next *right* event, and vice versa. It does this by suspending/resuming the corresponding iteration strand (Figure 3). In conjunction with *mostRecently*, we ensure that paired up events are forgotten. The result is the familiar correlation behavior of *zip*. Supplied effect handlers execute in right-to-left order. For example, the restrictions imposed by (*aligning*) apply first to ensure lockstep processing.

We can flexibly change the behavior of correlations with additional effect handlers. For instance, *event consumption* is another aspect we may wish to control. By default, there is no bound on how often a correlation may combine an observed event with another one. For example, in *combineLatest*, if *left* emits just one event e, then e will forever be combined with all future events from *right*. We may wish to enforce *affine* use of events to avoid this. It suffices to supply the *affinely* effect handler, just as in the *affine_latest* combinator in Figure 2, bottom right. Its behavior is depicted in Figure 1d, i.e., each event of the top stream occurs at most once in a pairing with the bottom stream.

2.5 Natural Specifications with Implicit Time Data

Notice that variable bindings in correlation patterns directly project the *payload* of events. For instance, the definition of *even_mouse_product* directly accesses the first component of p. As mentioned, the events actually carry additional time information: a closed *occurrence time interval* $[\tau_1, \tau_2]$. Yet, so far, time does not show up neither in the correlation pattern definitions nor in the types of the input reactives. This design choice enables programmers to write event relations and transformations naturally in terms of the payload, as long as they do not need explicit access to event times.

Cartesius provides access to event times through special variables. For each bound event variable x, Cartesius provides an implicit variable *?time*$_x$ that contains the time of x. The binding of *?time*$_x$ variables is managed internally by the join computation and exists only within the scope of the corresponding **correlate** pattern. For example, the pattern

```
correlate
 { p from mouse; k from keys
   where |(end ?time_p) − (end ?time_k)| ≤ 2 ms
   yield ⟨a,b⟩ }
```

correlates all mouse movements and keyboard events where the occurrence time intervals end at most 2 milliseconds apart from each other (Line 3). The function *end* in the constraint refers to the end of the respective interval. Dually, the function *start* refers to the beginning of the interval. When an event results from a correlation pattern, its time interval is the union of the intervals of the contributing events. With implicit variables, we avoid the notational overhead of wrapping and unwrapping the payload and time from *n* input events to form an output event.

2.6 Time Windows as Contextual Abstractions

Since effect handlers influence control flow, core features from the CQL-like query languages [Arasu et al. 2004; Krämer and Seeger 2009] and CEP engines [Demers et al. 2006; Diao et al. 2007] are expressible. One such core feature are *time windows*. The following example is adapted from EventJava [Eugster and Jayaram 2009] to find new TV releases having five good reviews within a month:

```
correlate
 { with (slidingWindow (1 Month) (1 Day))
     release     from TVReleases
     reviews(5) from TVReviews
     where (distinct reviews)
              (forall reviews (λx. (rating x) ≥ 3.5))
              (forall reviews (λx. (model x) == (model release)))
     yield release }
```

In the example, the pattern body is surrounded by the **with** syntactic form. As opposed to the implicit variable *?restriction*, which *injects* effect handlers into the middle of the pattern computation (Figure 3), the syntactic form **with** *encloses* the pattern computation with the given handler. This way, the entirety of the correlation pattern computation can be controlled. In our example, the *slidingWindow* handler bound by **with** manages multiple instances ("windows") of the correlation pattern: a new instance each day, processing events within the past month. This way, the *slidingWindow* handler emulates the design of stream query languages – windows impose a temporal scope in which the query executes.

2.7 Summary

In summary, the injection mechanism via *?restriction* is the main extension point for changing the correlation behavior. Importantly, this mechanism avoids unclear and hard-coded correlation parameters that numerous CEP and stream query languages exhibit. We support scoped variations of the correlation semantics at different places in a program.

3 CORE LANGUAGE AND DATA TYPES

This section defines the formal syntax and semantics of a core language for effects and handlers, λ_{cart}. In Section 4, we will define the semantics of Cartesius' high level correlation patterns (Section 2) by a translation into λ_{cart}.

The core language λ_{cart} is based on the second-order call-by-value λ-calculus, extended with algebraic data types, pattern matching and recursion. In addition, it features native support for algebraic effects, effect handlers and row-based effect polymorphism. λ_{cart} is similar to Koka [Leijen 2017b], whereas the presentation borrows heavily from Biernacki et al. [2018]. That is, we employ explicit type/effect abstraction and subtyping.

Expressions

$$
\begin{array}{llll}
v ::= \lambda x.e \mid \Lambda\alpha^{\kappa}.e \mid \mathsf{k}\ \overline{v} \mid com & \textit{(Val)} & x, y, z \ldots \ \textit{(Var)} & h \ ::= x \mapsto e \mid h;\ com\ x\ x \mapsto e \ \textit{(HCls)} \\
e ::= x \mid v \mid e\ e \mid e\ [T^{\kappa}] \mid \mathsf{k}\ \overline{e} \mid \textbf{fix}\ e & \textit{(Exp)} & \alpha^{\kappa}, \beta^{\kappa}, \gamma^{\kappa} \ldots \ \textit{(TVar)} & com \ ::= \cdots \ \textit{(Cmd)} \\
\quad \mid \textbf{match}\ e\ \{\overline{p \mapsto e}\} \mid \textbf{handle}\{h\}\ e & & p ::= x \mid \mathsf{k}\ \overline{p} \ \textit{(Pat)} &
\end{array}
$$

Types

$$
\begin{array}{llll}
T, U, V ::= \alpha^{*} \mid T \to^{\varepsilon} T \mid D\ \overline{T} \mid \forall \alpha^{\kappa}.T & \textit{(Typ)} & \kappa ::= * \mid \mathsf{e} \ \textit{(Kind)} & \Gamma ::= \varnothing \mid \Gamma, x : T \mid \Gamma, \alpha^{\kappa} \ \textit{(TCtx)} \\
D ::= \mathsf{Unit} \mid \mathsf{Nat} \mid \mathsf{Bool} \mid \mathsf{List}[T] \mid \cdots & \textit{(Data)} & T^{*} ::= T \ \textit{(TStar)} & \\
\varepsilon ::= \alpha^{\mathsf{e}} \mid \langle\rangle \mid \langle com, \varepsilon\rangle & \textit{(Row)} & T^{\mathsf{e}} ::= \varepsilon \ \textit{(TRow)} &
\end{array}
$$

Fig. 4. Syntax of λ_{cart}.

Evaluation Contexts

$$
\begin{array}{ll}
\mathcal{E} ::= [\cdot] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathcal{E}\ [T^{\kappa}] \mid \mathsf{k}\ \overline{v}\ \mathcal{E}\ \overline{e} \mid \textbf{fix}\ \mathcal{E} \ \textit{(ECtx)} & \mathcal{X}_{com} ::= [\cdot] \mid \mathcal{X}\ e \mid v\ \mathcal{X} \mid \mathcal{X}\ [T^{\kappa}] \mid \mathsf{k}\ \overline{v}\ \mathcal{X}\ \overline{e} \mid \textbf{fix}\ \mathcal{X} \ \textit{(XCtx)} \\
\quad \mid \textbf{match}\ \mathcal{E}\ \{\overline{p \mapsto e}\} \mid \textbf{handle}\ \{h\}\ \mathcal{E} & \quad \mid \textbf{match}\ \mathcal{X}\ \{\overline{p \mapsto e}\} \mid \textbf{handle}\ \{h\}\ \mathcal{X}, \\
 & \qquad \text{where } com \notin C(h)
\end{array}
$$

Handler Capabilities $C(h)$: $\quad C(x \mapsto e) = \{\} \qquad C(h;\ com\ x\ y \mapsto e) = \{com\} \cup C(h)$

Dynamics $\boxed{e_1 \longrightarrow e_2}$

$$
\begin{array}{rcll}
\mathcal{E}[(\lambda x.e)\ v] & \longrightarrow & \mathcal{E}[e\{v/x\}] & (\beta) \\
\mathcal{E}[(\Lambda\alpha^{\kappa}.e)\ [T^{\kappa}]] & \longrightarrow & \mathcal{E}[e[T^{\kappa}/\alpha^{\kappa}]] & (\textsc{TApp}) \\
\mathcal{E}[\textbf{match}\ v\ \{(p_i \mapsto e_i)_{1 \le i \le n}\}] & \longrightarrow & \mathcal{E}[e_j\sigma],\ \text{ where } v \Downarrow_{(p_i)_{1 \le i \le n}} \langle p_j, \sigma\rangle & (\textsc{match}) \\
\mathcal{E}[\textbf{fix}\ (\lambda x.e)] & \longrightarrow & \mathcal{E}[e\{\textbf{fix}\ (\lambda x.e)/x\}] & (\textsc{fix}) \\
\mathcal{E}[\textbf{handle}\ \{x \mapsto e; h\}\ v] & \longrightarrow & \mathcal{E}[e\{v/x\}] & (\textsc{ret}) \\
\mathcal{E}[\textbf{handle}\ \{h\}\ \mathcal{X}_{com}[com\ v]] & \longrightarrow & \mathcal{E}[e\{v/x, (\lambda y.\textbf{handle}\ \{h\}\ \mathcal{X}_{com}[y])/r\}], & (\textsc{handle}) \\
 & & \text{where } (com\ x\ r \mapsto e) \in h,\ \ y \text{ is fresh} &
\end{array}
$$

Pattern matching $\boxed{v \Downarrow_p \sigma \quad v \Downarrow_{\overline{p}} \langle p', \sigma\rangle}$

$$
v \Downarrow_x \{v/x\} \qquad \frac{(v_i \Downarrow_{p_i} \sigma_i)_{i \in \{1\ldots n\}}}{(\mathsf{k}\ v_1 \ldots v_n) \Downarrow_{(\mathsf{k}\ p_1 \ldots p_n)} \uplus_{i=1}^{n} \sigma_i} \qquad \frac{v \Downarrow_p \sigma}{v \Downarrow_{p\,\overline{p}'} \langle p, \sigma\rangle} \qquad \frac{\forall \sigma'. v \not\Downarrow_p \sigma' \qquad v \Downarrow_{\overline{p}'} \langle p'', \sigma\rangle}{v \Downarrow_{p\,\overline{p}'} \langle p'', \sigma\rangle}
$$

Fig. 5. Dynamic semantics of λ_{cart}.

We assume that algebraic data type signatures are pre-defined and well-formed, e.g.,

$$\textbf{type}\ \mathsf{List}[T] := \mathsf{nil} \mid \mathsf{cons}\ T\ \mathsf{List}[T]$$

is the type of lists. Examples of data values are: $\langle\rangle$ is the unit value of type Unit, true and false are of type Bool, $(\mathsf{cons}_T\ v\ \mathsf{nil}_T)$ is of type $\mathsf{List}[T]$ if v is of type T, $(\mathsf{S}\ (\mathsf{S}\ 0))$ is of type Nat and $\langle v_1, v_2\rangle$ is of pair type $\langle T_1, T_2\rangle$ if v_1 (resp. v_2) is of type T_1 (resp. T_2). For readability, we write numeric literals for Nat in the obvious way, and write list values in the usual bracket notation, e.g., $[0, 1] = \mathsf{cons}_{\mathsf{Nat}}\ 0\ (\mathsf{cons}_{\mathsf{Nat}}\ (\mathsf{S}\ 0)\ \mathsf{nil}_{\mathsf{Nat}})$.

Figure 4 shows the formal syntax of λ_{cart}, which for the most part is standard. We write $\mathsf{k}\ \overline{v}$ for applications of data constructors to values and correspondingly $D\ \overline{T}$ for instantiations of data types, in a fashion similar to Lindley et al. [2017]. We support polymorphism over both values and effect rows and hence annotate type variables with the kind $*$ or e, respectively. In examples, we sometimes omit the kind if it is unambiguous and we omit explicit type abstraction and application.

Figure 5 shows the operational semantics of λ_{cart} in terms of the reduction relation $e_1 \longrightarrow e_2$ on expressions, using evaluation contexts [Felleisen and Hieb 1992]. The rules (β), (Tapp), (match) and (fix) are standard, governing function application, type application, pattern matching and recursion, respectively. For brevity, the remainder of this section focuses on effects and dynamic semantics. We refer to Appendix A for the full static semantics.

3.1 Algebraic Effects and Handlers

Algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009] enable structured programming with user-defined effects in pure functional languages. Compared to more established language abstractions for effects, i.e., monads [Moggi 1991; Wadler 1992] and monad transformers [Liang et al. 1995], algebraic effects and handlers compose more freely and conveniently, because they support modular instantiation and modular abstraction via effect interfaces [Kammar et al. 2013]. Semantically, algebraic effects can be modeled in terms of free monads [Kiselyov and Ishii 2015; Swierstra 2008]. However, in this work, we treat effects and handlers as language primitives. Intuitively, handlers and effect commands are generalizations of **try**/**catch**/**throw** for managed exceptions. The difference is that handlers can resume evaluation at the point where the effect (e.g., thrown exception) occurred.

Correspondingly, λ_{cart} specifies the syntactic sort *com* for a set of *commands* (Figure 4), designating effectful operations. We assume that each command *com* has a predefined type $com\colon T_1 \longrightarrow T_2$, i.e., commands are functions from the client's perspective. We write concrete commands in bold blue font, e.g., $\textbf{yield}\colon \alpha \longrightarrow \text{Unit}$.

An effect handler h is a finite sequence of clauses, with one mandatory return clause $x \mapsto e$ and optional command clauses $com\ x\ r \mapsto e$, specifying how to handle/interpret a specific selection of commands. We always assume that for each command there is at most one corresponding clause in each handler. Handlers are second-class and applied to computations invoking effects using the **handle** $\{h\}\ e$ form. Intuitively, a handler h is a computation transformer $T_1\ {}^{\langle\overline{com}\rangle}\!\!\Rightarrow^{\varepsilon} T_2$, turning a computation of result type T_1 and effects $\langle\overline{com}\rangle$ (handled by h's clauses) to a computation with result T_2 and new effects ε (cf. Appendix A).

Reduction rules (RET) and (HANDLE) (Figure 5) govern the behavior of handler application. The former rule applies the return clause[2] of the handler for transforming the final result of the computation. The latter rule specifies how to handle effects invoked by the computation. Similarly to managed exceptions, a command invocation $com\ v$ shifts the control flow to the currently innermost handler with the capability to handle *com*. Just as Leijen [2017b], we express this by restricting the evaluation context $\mathcal{X}_{com}$, i.e., evaluation may focus under a handler application only if h does not handle *com*.

Once control flow shifts into a handler clause $com\ x\ r \mapsto e$, the argument to the command is bound to the first variable x and the *resumption* of the computation is bound to the second variable r. That is, e has the capability to resume the command invocation with an answer value. Hence, effects and handlers implement a more structured form of delimited continuation [Bauer and Pretnar 2015; Forster et al. 2017; Kammar et al. 2013]. Note that λ_{cart} employs *deep handlers*, i.e., the resumption re-applies the current handler to the rest of the computation, reflecting the intuition that handlers are folds over computation trees [Lindley 2014].

3.1.1 Effect Typing. We assume a row-based type and effect system as in Koka, which assigns effect rows to arrow types. For example, $map\colon \forall \alpha\, \beta\, \mu.(\alpha \longrightarrow^{\langle\mu\rangle} \beta) \longrightarrow^{\langle\rangle} \text{List}[\alpha] \longrightarrow^{\langle\mu\rangle} \text{List}[\beta]$ is effect polymorphic, indicated by the universally quantified effect variable μ. Because *map* applies the supplied function elementwise to its second argument, it overall induces the same effects μ. The order of effects in rows does not matter, e.g., $\langle \textbf{yield}, \textbf{fail}, \mu\rangle$ is equivalent to $\langle \textbf{fail}, \textbf{yield}, \mu\rangle$, which is ensured by subtyping and subsumption (Appendix A).

For simplicity, in contrast to Koka, we do not group commands into effect interfaces. Instead, each command induces itself as effect, e.g., if $com\colon T_1 \longrightarrow T_2$ is the predefined signature of *com*, then $T_1 \longrightarrow^{\langle com\rangle} T_2$ is the type assigned to *com* at the level of expressions. However, in examples we

[2] In examples, we omit the return clause, if it is an identity $x \mapsto x$ and omit curly braces in favor of indentation.

Derived Syntax

$\{T\}^{\varepsilon} \rightsquigarrow (\mathsf{Unit} \to^{\varepsilon} T)$	*(Thunk Typ)*	$\mathbf{let\ implicit}\ ?var\colon T = e_1\ \mathbf{in}\ e_2$	*(Implicit Def)*
$\{e\} \rightsquigarrow \lambda x.e,\ \text{ where } x \notin \mathrm{fv}(e)$	*(Thunk)*	$\rightsquigarrow \mathbf{let}\ x = e_1\ \mathbf{in}\ (\mathbf{handle}\ \{\mathbf{var}\ y\ k \mapsto k\ x\}\ e_2),$	
$\mathbf{handler}\{h\} \rightsquigarrow \lambda thnk.\mathbf{handle}\ \{h\}\ (thnk\ \langle\rangle)$	*(Handler Val)*	where x, y, k are fresh and $\mathbf{var}\colon \mathsf{Unit} \to T$	
$\mathbf{with}\ e_1\ e_2 \rightsquigarrow e_1\ \{e_2\}$	*(With)*	$?var \rightsquigarrow \mathbf{var}\ \langle\rangle$	*(Implicit Use)*

Handler Combinators

$$\_ \boxplus \_ := \lambda hl_1.\lambda hl_2.\lambda thnk.\mathbf{with}\ hl_1\ \mathbf{with}\ hl_2\ (thnk\ \langle\rangle) \qquad \textit{(Handler Composition)}$$

Fig. 6. Derived syntax and combinators.

```
let accum = λn. handler(s = n)
  x ↦ s
  yield val resume ↦
    resume ⟨⟩ (s + val)
```

```
let interactive = handler
  yield val resume ↦
    println val;           //print next element to console
    match readchar ⟨⟩      //read keyboard
      \cr ↦ resume ⟨⟩      //proceed on carriage return
      _   ↦ ⟨⟩             //otherwise, terminate
```

Fig. 7. Different Interpretations of **yield**.

sometimes abbreviate sets of commands in effect rows by a single effect name. We further allow polymorphic commands, writing type parameters and instantiation as subscripts, e.g., $\mathbf{return}_{\alpha}\colon \alpha \longrightarrow \mathsf{Unit}$.

3.1.2 First Class Handlers and Combinators. Recall from Section 2 that we frequently use (1) combinators to construct restriction handlers as *values*, (2) handler composition ⊞ and (3) implicit variables for injecting handlers. Accordingly, we define convenience syntax for first class handlers and thunks in terms of functions and second class handlers (Figure 6). Encoding first class handlers in this way keeps the core language simple, i.e., handler values become thunk-accepting functions and handler application becomes function application. Correspondingly, if a second class handler h has type $T_1 \overset{\langle\overline{com}\rangle}{\Rightarrow}{}^{\varepsilon}\ T_2$, then its first class encoding $\mathbf{handler}\{h\}$ has type $\{T_1\}^{\langle\overline{com},\varepsilon\rangle} \longrightarrow^{\varepsilon} T_2$ (cf. Appendix A).

To clearly convey the intent (and for aesthetic reasons), we use Eff-style **with** syntax [Bauer and Pretnar 2015] for applying handler values. Composing handlers (⊞) simply becomes nested handler application.

We further allow the usual **let** and **letrec** binding forms, sequencing $e_1; e_2$ as well as *parameterized handlers* [Kammar et al. 2013; Plotkin and Pretnar 2009] in a notation similar to Koka, i.e., $\mathbf{handler}(x = e)\{h\}$, binding the result value of e to a variable x, which is accessible from all clauses of handler h. Due to space limitations, we elide the desugaring of parameterized handlers and refer to Leijen [2017b]. One may think of them as pure functions accepting the parameter value and yielding a first class handler.

3.1.3 Example. In the following example, we consider the command **yield** : Nat ⟶ Unit and the computation $c := map\ (\mathbf{yield})\ [1, 2, 3, 4]$, where map is the standard mapping function on lists. We may turn this mapping computation into an accumulating computation, *without changing* c, by enclosing the computation with an appropriate handler for **yield**, e.g., *accum* in Figure 7. This handler sums up the yielded values in its handler parameter and would take the following high-level evaluation steps:[3]

$$\begin{aligned}
&\mathbf{with}\,(accum\ 0)\ (map\ (\mathbf{yield})\ [1,2,3,4]) \longrightarrow^{*} \mathbf{with}\,(accum\ 0)\ (\mathrm{cons}\ (\mathbf{yield}\ 1)\ (map\ (\mathbf{yield})\ [2,3,4]))\\
\longrightarrow^{*}\ &\mathbf{with}\,(accum\ 1)\ (\mathrm{cons}\ \langle\rangle\ (map\ (\mathbf{yield})\ [2,3,4])) \longrightarrow^{*} \mathbf{with}\,(accum\ 3)\ (\mathrm{cons}\ \langle\rangle\ (\mathrm{cons}\ \langle\rangle\ (map\ (\mathbf{yield})\ [3,4])))\\
\longrightarrow^{*}\ &\mathbf{with}\,(accum\ 10)\ [\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle] \longrightarrow 10.
\end{aligned}$$

[3] We leave a proper reduction trace as an exercise to the reader.

Syntax

$$\begin{array}{llll} \tau \in \mathcal{T} & \textit{(TStamp)} & v ::= \ldots \mid b & \textit{(Val)} \\ b ::= \tau \mid \bot \mid \top & \textit{(TBound)} & e ::= \ldots \mid e \sqcup e \mid e \leq e & \textit{(Exp)} \end{array}$$

Dynamics

$$\frac{b_1 \preceq b_2}{b_1 \leq b_2 \longrightarrow \mathsf{true}}\ (\leq\text{T}) \qquad \frac{b_1 \npreceq b_2}{b_1 \leq b_2 \longrightarrow \mathsf{false}}\ (\leq\text{F}) \qquad \frac{\tau_1'' = \tau_1 \sqcap \tau_1' \qquad \tau_2'' = \tau_2 \sqcup \tau_2'}{\langle \tau_1, \tau_2 \rangle \sqcup \langle \tau_1', \tau_2' \rangle \longrightarrow \langle \tau_1'', \tau_2'' \rangle}\ (\sqcup)$$

Fig. 8. Time stamps/interval syntax and semantics.

Note that the resumption of a parameterized handler now takes an extra argument, which is the new value of the handler parameter (Line 4 of *accum*).

As the previous example shows, handlers define the meaning to commands, such as **yield**. We can easily give it a vastly different meaning by applying another handler, e.g., *interactive* in Figure 7. This handler makes c print its elements to the console, where its progression is controlled by the user's keyboard interactions. The choice to progress has "flipped" from *internal* to *external* choice, thanks to handler clauses exposing the resumption. That is, effects and handlers support coroutining [de Moura and Ierusalimschy 2009].

3.1.4 Implicit Variables as Effects. We encode implicit variables and bindings from Section 2 in terms of effects (Figure 6, rules *(Implicit Def)* and *(Implicit Use)*). For each implicit variable declaration *?var* we associate a command **var**: Unit $\longrightarrow T$ (without question mark). Clients (i.e., e_2 in *(Implicit Def)*) invoke this command to retrieve the bound value, where an occurrence *?var* desugars into the command invocation **var** $\langle\rangle$ by rule *(Implicit Use)*. Note that effect typing naturally enables static tracking of implicit dependencies, e.g., the effect row assigned to e_2 would have the shape $\langle$**var**$, \varepsilon\rangle$, mentioning the occurrence of $?var$. This form of implicit variables resembles the *dynamic scoping with static types* work by Lewis et al. [2000], since command dispatch and hence accessing the implicit value is dynamic. For static implicits, as in Haskell or Scala, we would require coeffects [Petricek et al. 2014].

3.1.5 The Role of Handlers in Cartesius. We can think of commands as effect constructors in computations. Dually, handlers deconstruct/observe effects as the computation unravels, i.e., they are *co-algebraic*. Later, in Section 4, we exploit this intuition to encode event notifications as effects and event observers as handlers. Yet, this is not the only use of handlers: Similarly to the *interactive* example above, handlers enable us to "flip" from direct style to callback style, transparently. Last, but not least, handlers realize restrictions (Section 2.4) of the cartesian product in the sense of Proposition 1.2. That is, effects occurring in the cartesian product computation can be locally reinterpreted by handlers, changing how the computation behaves.

3.2 Time and Event Values

We assume a discrete time model to track the arrival times of event notifications (Figure 8). The set $\mathcal{T}$ is an infinite set of discrete, totally ordered *time stamps* τ. In timing predicates, we lift time stamps to *time bounds* b, extending $\mathcal{T}$ to a complete lattice with a least element $\bot$ ("minus infinity"), a greatest element $\top$ ("plus infinity"), and greatest lower bound ($\sqcap$) as well as least upper bound ($\sqcup$) operations. We overload $\preceq$ to denote the lifted order on time bounds.

Recall that an *event value* is evidence of a past situation of interest (Section 2.1). It is either supplied externally or the result of joining one to many other event values. Event values have type **type** Ev$[T]$:= ev T $\langle$Time, Time$\rangle$, where the first component is the *content/payload* and the second component is the *occurrence time interval* $\langle\tau_1, \tau_2\rangle$ in pair notation. For each external event, we assume that the runtime assigns a discrete time interval containing only the event's arrival time. Joining n event values (ev v_1 i_1), ..., (ev v_n i_n) results in an event value (ev $f(v_1, \ldots, v_n)$ $(i_1 \sqcup \ldots \sqcup i_n)$),

```
1 let rec $eat$: $\forall\alpha\,T.\mathsf{R}[T] \rightarrow^{\langle\rangle} (\mathsf{Ev}[T] \rightarrow^{\langle\alpha\rangle} \mathsf{Unit}) \rightarrow^{\langle\mathbf{async},\alpha\rangle} \mathsf{Unit} = \Lambda\alpha.\Lambda T.\lambda r.\lambda f.$
2   match ($\mathbf{await}_T\ r$)
3     rcons $hd\ tl$ $\mapsto (f\ hd);\ eat_{\alpha,T}\ tl\,f$
4     rnil          $\mapsto \langle\rangle$
```

Fig. 9. Direct style, asynchronous iteration.

which merges the payloads via a function f and the time intervals with the $\sqcup$ operator, yielding the smallest interval containing all given intervals. In Cartesius, the merge function f corresponds to the **yield** expression in patterns (Section 2.2). We discuss it further in Section 4.

3.3 Asynchrony and Push/Pull Reactives

Since event sources are asynchronous, we must commit to a push-based processing model. Inadvertently, this leads to a programming style where control is inverted, requiring difficult to manage callbacks ("callback hell" [Edwards 2009]). We already demonstrated by the *interactive* example above how effect handlers reconcile inversion of control with direct style, iterative programs. What remains to be shown is how reactives from Section 2.1 integrate with this programming pattern.

In λ_{cart}, reactives $\mathsf{R}[T]$ correspond to a co-inductive list type, which is inspired by Elliott [2009]:

$$\textbf{type}\ \mathsf{R}[T] := \mathsf{Future}[\mathsf{R}'[T]] \qquad \textbf{type}\ \mathsf{R}'[T] := \mathsf{rnil} \mid \mathsf{rcons}\ \mathsf{Ev}[T]\ \mathsf{R}[T].$$

The data type $\mathsf{Future}[T]$ represents async/await style futures [Bierman et al. 2012; Haller and Miller 2013; Syme et al. 2011], which implement the potential $\Diamond[T]$ of an event source to yield its next event value. Futures can be implemented on top of algebraic effects [Dolan et al. 2017; Leijen 2017a], but we elide the definition due to space limitations, treating $\mathsf{Future}[T]$ as abstract. The only way to introduce and eliminate future values is via effects. Command $\mathbf{async}_{\alpha,\mu}$:$\{\alpha\}^{\langle\mu\rangle} \rightarrow \mathsf{Future}[\alpha]$ immediately returns a future to the caller, asynchronously executing the given thunk. The elimination command $\mathbf{await}_\alpha$:$\mathsf{Future}[\alpha] \rightarrow \alpha$ returns the result value of a completed future, if it is completed and otherwise supends the caller. In effect rows, we simply abbreviate all the effects associated with asynchrony as **async**. We assume that at the top-level, there is always a handler present for handling the **async** effects.

Since $\mathsf{R}[T]$ resembles a (possibly infinite) list, we can now write direct style iterative programs over the events originating from an event source, in a fashion similar to the *interactive* example. For this purpose, we define the polymorphic iteration combinator *eat* (Figure 9), which applies a function elementwise to all events of a given reactive. The positions of $\mathsf{R}[T]$ wrapped in $\mathsf{Future}[\cdot]$ precisely mark where internal choice (pull) may "flip" to external choice (push). That happens if the next event is not yet available (Line 2 of *eat*). Thus, we have raised the level of abstraction and do not need to worry about low level callback functions.

3.4 Interleaving

We conclude this section with one more abstraction for asynchrony. Recall from Figure 3, that correlation patterns require concurrent executions of threads/strands. One can define a combinator for that, called *interleave*, on top of the asynchrony system above. Here is its simplified signature:

$$interleave\colon \forall\mu.\mathsf{List}[\{\mathsf{Unit}\}^{\langle\mathbf{async},\mu\rangle}] \rightarrow^{\langle\mathbf{async},\mu\rangle} \mathsf{Unit}.$$

The combinator concurrently executes a list of independent, asynchronous computation strands, passed as thunks. Due to space limitations, we elide its definition and refer for the details to Leijen [2017a]. Moreover, we assume that the scheduling of the concurrent execution is fair and allow the syntax $e_1 \parallel \ldots \parallel e_n \rightsquigarrow interleave\ [e_1, \ldots, e_n]$ for readability.

The above type is noteworthy: The effect polymorphism in *interleave* states that all the effects occurring in the given asynchronous strands *are observable by the caller*. For example,

$$\{\mathbf{await}_T\ x\} \parallel \{\mathbf{println}\ \text{"}\mathit{foo}\text{"}\} \parallel \{\mathbf{yield}\ 1\}$$

induces the effects $\langle$**async**, **println**, **yield**$\rangle$ to the context in which it is invoked.

This is an important property of the interleaving combinator, which we exploit for the implementation of Cartesius in the next section.

4 EVENT CORRELATION WITH ALGEBRAIC EFFECTS

In Section 3, we defined required preliminaries for Cartesius in terms of λ_{cart}, from effects and handlers to event values, reactives and interleaving. In this section, we present the implementation of Cartesius from Section 2 as a shallow embedding in λ_{cart}. Our design follows the computational interpretation of joins (Proposition 1.2 and Figure 3). In particular, while Section 2 motivated Cartesius from the perspective of end users, this section explains the overall framework that enables library writers to define new, custom join behaviors.

We prototyped Cartesius in two languages with native support for algebraic effecs, Koka [Leijen 2017b] and multicore OCaml [Dolan et al. 2017]. The implementations are available at:

http://github.com/bracevac/cartesius

4.1 The Marriage of Effects and Joins: Correlate by Handling

We start with motivating the underlying structure of correlation computations in Cartesius (depicted in Figure 3), which yields a useful, modular organization principle in terms of effects and handlers.

One can straightforwardly encode event correlation behavior over $n \geq 1$ reactives by *nested iteration*. Recall that in Section 3, we introduced abstractions for direct style iteration over asynchronous reactives. In particular, we defined the higher order iteration function *eat* (Figure 9), which "feeds" observed events one-by-one to its function argument f. Thus, nested iteration takes the form

$$\mathit{eat}\ r_1\ (\lambda ev_1.\mathit{eat}\ r_2\ (\lambda ev_2.\ \cdots\ \mathit{eat}\ r_n\ (\lambda ev_n.\ \mathit{process}\ \langle ev_1, \ldots, ev_n\rangle)\cdots))$$

for reactives r_1:$\mathsf{R}[T_1]$,..., r_n:$\mathsf{R}[T_n]$ and a function *process*: $(\mathsf{Ev}[T_1], \ldots, \mathsf{Ev}[T_n]) \rightarrow \mathsf{Unit}$, which is a callback on completed n-tuples, encapsulating a concrete event correlation behavior. The nesting enforces a sequential selection of events, i.e., before the computation binds event ev_i, it must bind (and **await**) all events ev_j, $j < i$.

This programming pattern is not uncommon and can be found in similar contexts, e.g., language-embedded query APIs and comprehension notations based on monads [Cheney et al. 2013; Meijer et al. 2006; Wadler 1990a]. In these APIs, multiple selections translate into nested applications of the well-known combinator *bind*:[4] $\mathsf{M}[A] \rightarrow (A \rightarrow \mathsf{M}[B]) \rightarrow \mathsf{M}[B]$ on a monad $\mathsf{M}[\cdot]$ (cf. this LINQ-based Rx.NET Example [2013] correlating events using nesting).

Unfortunately, such nesting can *not* faithfully model all kinds of joins over asynchronously arriving events, because of the induced sequential selection order. In event correlation, reactives produce events in an arbitrary order. For example, the *combineLatest* combinator from Section 2 must continue to process events from one reactive, even if the other has stopped producing events.

[4] *bind* is also known as *SelectMany* and *flatMap*.

4.1.1 Interleaved Binding. The problem is that the (static) notation order of event bindings determines the dynamic selection order during event correlation, while, in general, the two should be independent. One solution is to extend monads with new combinators and laws, e.g., Joinads [Petricek et al. 2011; Petricek and Syme 2011]. However, the same drawbacks to using monads apply, which we mention in Section 3.1. In this work, we use algebraic effects and handlers for decoupling the (static) notation order of event bindings from the dynamic selection order during correlation. Instead of nesting the n iterations, we *juxtapose* them, by using the *interleave* combinator (Section 3.4):

$$\{eat\ r_1\ \lambda ev_1.e_1\} \parallel \cdots \parallel \{eat\ r_n\ \lambda ev_n.e_n\}.$$

This composition puts each iteration in a separate computation strand. No event binding takes precedence over the other, all iterations proceed independently and concurrently. Our solution seamlessly fits in our setting, where algebraic effects and handlers are a key element of the design. As a side-effect, the solution also represents a novel usage of algebraic effects and handlers.

4.1.2 Correlate by Handling. How can the iterations be correlated, if the interleaving separates them? They can be correlated by effect handling! We simply follow the example in Section 3.1.3, applying it to reactives instead of lists. That is, we elementwise invoke effect commands:

$$\{eat\ r_1\ \lambda ev_1.\mathbf{push}_1\ e_1\} \parallel \cdots \parallel \{eat\ r_n\ \lambda ev_n.\mathbf{push}_n\ e_n\},$$

which after η-conversion becomes:

$$\{eat\ r_1\ (\mathbf{push}_1)\} \parallel \cdots \parallel \{eat\ r_n\ (\mathbf{push}_n)\}. \tag{1}$$

We let each iteration in the interleaving invoke a distinct, *fresh* effect $\mathbf{push}_i\colon \mathsf{Ev}[T_i] \longrightarrow \mathsf{Unit}$. Its purpose is to *expose each produced event as an effect.* The effect name and signature are position dependent, so that at the type and term level, one can discern which input a **push** belongs to.

Importantly, the effect type of the interleaving (1) is revealing the nature of event correlation. Recall from Section 3.4 that the effects occurring in the interleaved strands propagate to the calling evaluation context. Thus, the effect type of expression (1) is

$$\langle \mathbf{push}_1, \ldots, \mathbf{push}_n, \mathbf{async} \rangle,$$

which intuitively states that at any time, in any order and arbitrarily often, events from the n reactives may "pop into existence" as effect invocations. That concisely characterizes the problem of asynchronous event correlation!

Moreover, the effect type hints at how to correlate events from n reactives. The interleaving is namely the ideal place to implement event correlation, since it is where all events are exposed as effects to the calling context. That is, we enclose the interleaving (1) with effect handlers and *correlate by handling*:

Definition 4.1 (Handler-based Event Correlation). An asynchronous n-way join computation over reactives r_1 to r_n is a composition of effect handlers $h_1 \boxplus \cdots \boxplus h_k$ enclosing the interleaved iteration

$$\mathbf{with}\ (h_1 \boxplus \cdots \boxplus h_k)\ ((\{eat\ r_1\ (\mathbf{push}_1)\} \parallel \cdots \parallel \{eat\ r_n\ (\mathbf{push}_n)\})\ \langle\rangle),$$

so that all the $\mathbf{push}_i$ are *discharged.* That is, its type has the form

$$(h_1 \boxplus \cdots \boxplus h_k)\colon \{\mathsf{Unit}\}^{\langle \mathbf{push}_1, \ldots, \mathbf{push}_n, \mathbf{async}, \varepsilon \rangle} \longrightarrow^{\langle \mathbf{async}, \varepsilon \rangle} \mathsf{Unit}. \qquad \blacksquare$$

The above definition yields a useful, modular organization principle for event correlation computations. Interleaved iteration can be defined once and for all, as a generic component (Figure 3) and there is a clearly defined structure to computations. All event occurrences "join" in the context of the interleaving, where they can be correlated by handling. Effect handlers are modular units of composition. They interpret underlying event notifications, embodied by the family of **push** effects, which form an interface. The implementation effort for event correlation reduces to programming handlers against that interface.

```
let join_shape_n = {
  with trigger_n(?pattern) // test against pattern
    with memory_n          // memory state
      with reify_n         // cartesian product
        with ?restriction_n // inject restrictions
          with forAll_n    // observe and store events
            ?streams_n ⟨⟩ } // inject stream iteration
```

```
let reify_n = handler
  {push_i ev resume ↦
    let entry = find ev (get_i ⟨⟩) in
      //fire the join pattern with all tuples in memory:
      forEach (trigger_n) (cartesian_i entry);
      GC ⟨⟩; //delete memory entries having zero lifespan
      resume ⟨⟩}_{1≤i≤n}
```

```
let forAll_n = handler
  {push_i ev resume ↦
    set_i ((ev, inf) :: (get_i ⟨⟩));
    resume (push_i ev)}_{1≤i≤n}
```

```
let memory_n =
  ⊞_{i=1}^n handler(s: List[Ev[T_i] × Count] = nil)
          get_i ⟨⟩ resume ↦ resume s s
          set_i s′ resume ↦ resume ⟨⟩ s′
```

```
type Body_n = (Ev[T_1] × . . . × Ev[T_n]) →^⟨yield,fail⟩ Unit
let trigger_n = handler(pattern: Body_n)
  trigger_n tuple resume ↦ pattern tuple; resume ⟨⟩ pattern
```

Fig. 10. Basic shape of n-way join computations.

Cartesius is all about interpreting interleavings of **push** effects with a suitable handler context ($h_1 \boxplus \cdots \boxplus h_k$). In the remainder of this section, we will define an encoding of correlation patterns (Figure 3) and the constrained cartesian product (Proposition 1.2) in terms of such a handler context.

4.2 Core Framework of Cartesius

Now we make use of Definition 4.1 and encode a generic, extensible cartesian product computation (Section 2.4 and Figure 3) in terms of handlers.

4.2.1 Interface. Our encoding requires the following signature of effect commands

$$\mathbf{push}_i\colon \mathsf{Ev}[T_i] \longrightarrow \mathsf{Unit},\ 1 \le i \le n \qquad \mathbf{get}_i\colon \mathsf{Unit} \longrightarrow \mathsf{List}[\mathsf{Ev}[T_i] \times \mathsf{Count}],\ 1 \le i \le n$$
$$\mathbf{trigger}_n\colon (\mathsf{Ev}[T_1] \times \ldots \times \mathsf{Ev}[T_n]) \longrightarrow \mathsf{Unit} \qquad \mathbf{set}_i\colon \mathsf{List}[\mathsf{Ev}[T_i] \times \mathsf{Count}] \longrightarrow \mathsf{Unit},\ 1 \le i \le n$$

which is indexed by the number n of input reactives and their event types $T_1, \ldots, T_n$.

Next to the $\mathbf{push}_i$ commands for event notifications from Section 4.1, there are $\mathbf{get}_i/\mathbf{set}_i$ commands for reading/writing a local memory ("mailbox"), one for each input. This is the effect interface to the local, shared memory component in Figure 3, which retains event notifications for processing. Each memory is an association list, storing currently relevant event values along with their *lifespan* of **type** Count := fin Nat | inf. This type specifies how often (finitely or arbitrarily often) an in-memory event value may be reused in combination with other event values. By default, in-memory event values have infinite lifetime (inf). This policy may be changed by restriction handlers, e.g., affine events (Section 2.4) would have lifespan (fin 1). Once a correlation computation materializes a candidate n tuple, it is passed to the command $\mathbf{trigger}_n$, for testing against the **where** predicate (Figure 3).

4.2.2 Generative Effects. Above, we specified an *indexed family of effects* to express that each instantiation of a join computation uses different, instance-specific version of the effects **push**, **get**, **set**, and **trigger**. Such definitions are known as *generative effects*. These kinds of effects are still an open research problem and not readily supported in current language implementations. For simplicity, we assume a sufficient supply of predefined, differently named commands. We further discuss in Section 7 how we approximated generative effects in our implementations.

4.2.3 Implementation. Here, we define the generic, extensible n-way cartesian product implementation, in terms of handlers for the $\mathbf{join}_n$ signature above. The implementation is represented by the computation thunk *join_shape*$_n$ in Figure 10, which composes (i.e., layers) a number of sub-handlers. These sub-handlers are in close correspondence to the schematic boxes inside **correlate** in Figure 3, in upside down order. In the following, we explain the individual handlers comprising this shape, bottom to top.

At the bottom, there is the implicit variable (cf. Section 3.1.4)

$$?streams_n \colon \{\mathsf{Unit}\}^{\langle \mathbf{push}_1, \ldots, \mathbf{push}_n, \mathbf{async} \rangle},$$

which accepts any computation invoking the **push** effects for correlation, as an external dependency. Its is by default bound to the interleaved iterations from Definition 4.1. The binding can be overridden externally, e.g., our encoding of time windows requires binding a different computation (cf. Section 4.5).

The *forAll*$_n$ handler in *join_shape*$_n$ reacts to each **push**$_i$ command by adding the supplied event to the i-th memory with an initially unbounded lifespan. Note that the *resume* invocation at the end implements a coroutine behavior (Section 3.1) for the i-th iteration and forwards the current **push**$_i$ command further up the context, to give other handlers the chance to process the command invocation. Iteration is continued as soon as one layer in the stack resumes with the unit value, or it is stopped if a handler decides to not invoke the resumption.

The ?*restriction*$_n$ implicit variable in *join_shape*$_n$ is our main extension point for changing the behavior of the cartesian product by external injection of restriction handlers. We showed its usage in Section 2 and postpone how to program concrete restriction handlers to Section 4.4.

The *reify*$_n$ handler in *join_shape*$_n$ materializes the cartesian product over all in-memory event values having non-zero life time, each time a new **push**$_i$ event notification reaches this layer. It then invokes the **trigger**$_n$ command for each resulting n-tuple (see below). The function

$$cartesian_i \colon (\mathsf{Ev}[T_i] \times \mathsf{Count}) \longrightarrow \mathsf{List}[\mathsf{Ev}[T_1] \times \ldots \times \mathsf{Ev}[T_n]]$$

computes the candidate tuples from the just observed event *ev* and the memory contents for all inputs $j \neq i$. After triggering, used-up event values are garbage collected from the memory. This is the terminal coroutine layer for all **push**$_i$ commands, because it does not forward **push**$_i$ effect before resuming (Line 7 of *reify*$_n$).

The *memory*$_n$ handler in *join_shape*$_n$ maintains and threads the current state of the memory through the join computation. That is, it keeps the n-tuple of all n memory states in handler parameters and answers the **get**$_i$/**set**$_i$ commands (Section 4.2.1) by retrieving/updating the i-th memory.

Finally, the *trigger*$_n$ handler is responsible for testing each materialized candidate tuple that is propagated via the **trigger**$_n$ effect against the constraints in the correlation pattern, which is bound to the implicit variable ?*pattern* as part of the correlation pattern translation (see next Section). Once a candidate tuple satisfies the constraints, it is marked as consumed. That is, the life time counters of its n components are decreased, and it is inserted into the output reactive of the correlation computation.

4.3 Correlation Pattern Translation

Now, we translate correlation patterns (Section 2) into λ_{cart} (Section 3), in two steps: (1) desugar the body of patterns (the part enclosed by **correlate**$\{\ldots\}$) into λ_{cart} code and (2) instantiate a *join_shape*$_n$ computation (Section 4.2) with the desugared pattern.

Language-embedded comprehension syntax is well-understood for monadic APIs [Wadler 1990a], which translates into monad combinators. However, in this work, we present how comprehension syntax alternatively could be translated in terms of algebraic effect handlers.

```
1 $[\![ \{ (x_i \textbf{ from } e_i)_{1\le i\le n} \textbf{ where } (e_{p_i})_{1\le i\le k} \textbf{ yield } e_r \} ]\!] =$
2   { match (bind$_n$ $\langle e_1, \ldots, e_n\rangle$)
3     $\langle$ev $x_1$ $t_1, \ldots,$ ev $x_n$ $t_n\rangle \mapsto$
4       (let implicit $?time_{x_i}$ : Time = $t_i)_{1\le i\le n}$ in
5         if $(e_{p_1} \wedge \cdots \wedge e_{p_k})$
6         then yield (ev $e_r$ $(t_1 \sqcup \cdots \sqcup t_n)$)
7         else fail $\langle\rangle$ }
```

```
1 let $setup_n$ = handler
2   bind$_n$ $\langle r_1, \ldots, r_n\rangle$ $body \mapsto$
3     let implicit $?pattern$ = $body$ in
4     let implicit $?streams$ =
5       $\{\{eat\ r_1\ (\textbf{push}_1)\} \parallel \cdots \parallel \{eat\ r_n\ (\textbf{push}_n)\}\}$
6       in $join\_shape_n$ $\langle\rangle$
7 let $correlate_n$ = $\lambda body.$ with $(gen \boxplus setup_n)$ $(body\ \langle\rangle)$
```

Fig. 11. Correlation Pattern Translation. Left: Pattern Binders and Body. Right: **correlate** delimiter.

4.3.1 *Step 1: Pattern Body Translation.*

The left-hand part of Figure 11 shows the first translation step, producing code that invokes the following effects:

$$\textbf{yield}: \mathsf{Ev}[T_{n+1}] \longrightarrow \mathsf{Unit} \qquad \textbf{bind}_n: (\mathsf{R}[T_1] \times \ldots \times \mathsf{R}[T_n]) \longrightarrow (\mathsf{Ev}[T_1] \times \ldots \times \mathsf{Ev}[T_n])$$
$$\textbf{fail}: \mathsf{Unit} \longrightarrow \mathsf{Unit}$$

The **bind**$_n$ command declares that the correlation pattern extracts candidate n-tuples from n input reactives. Its purpose is mapping the variable bindings (x_i **from** e_i) to ordinary λ_{cart} bindings. That means, the right-hand sides of **from** are grouped into the parameter list of **bind**$_n$ (Line 2), which is invoked to extract an n-tuple from the inputs. The left-hand side variables of **from** are grouped into binders of the pattern matching clause (Line 3), which decomposes a supplied n-tuple. The **where** and **yield** clause of the pattern are transformed into an **if** expression. If all constraints are satisfied, then the **yield** effect is invoked with the result. Otherwise, **fail** is invoked. **yield** signals the run time that the resulting event should be appended to the output reactive of the correlation (cf. Figure 3).

Following Section 2.5, this desugaring separates the payloads of events from their intervals, so that programmers can write natural constraints and transformations. Accordingly, time intervals can be accessed via the implicit variable $?time_{x_i}$, which is another important use case of generative effects in Cartesius (cf. Section 4.2.2).

Finally, the argument expression to **yield** re-wraps the result, implementing the event joins from Section 3.2.

4.3.2 *Step 2: Instantiation.*

In the second translation step (Figure 11, right-hand part), the $setup_n$ handler stages an instance of the join shape computation (Figure 10), linking it with the previous translation step, using implicit variables. Note how Line 2 on both sides complement each other. The previous translation step invokes **bind**$_n$, which in turn is handled by $setup_n$. In this way, its resumption *body* captures Lines 2-7 of the translated pattern as a function into the implicit variable $?pattern$. Recall that $?pattern$ is invoked by the $trigger_n$ handler (Section 4.2.3) with each tuple materialized during event correlation.

Another responsibility of $setup_n$ is constructing and binding the interleaved iterations (Section 4.1.2) from the given reactives r_1 to r_n to $?streams_n$, so that the $join\_shape_n$ instance correlates its event notifications.

Finally, we encode the **correlate** delimiter simply as a λ-abstraction (Figure 11, right-hand part, Line 7), which applies $setup_n$ to its argument. The latter is the translation result from the first step.

The overall pattern translation proceeds as follows:

```
let $r_{out}$ = correlate { $(x_i$ from $e_i)_{1\le i\le n}$   ⤳  let $r_{out}$ =                                          ⤳  let $r_{out}$ =
               where $(e_{p_i})_{1\le i\le k}$                      let $\$b$ = $[\![ (x_i$ from $e_i)_{1\le i\le n} \cdots ]\!]$         let $\$b$ = { match bind$_n$ $\cdots$}
               yield $e_r$ }                                   in $correlate_n$ $\$b$ // $b is fresh, n is inferred        in with $(gen \boxplus setup_n)$ $(\$b\ \langle\rangle)$
```

```
let mostRecently_{n,i} = handler
  push_i ev resume ↦
    set_i (⟨ev, inf⟩ :: nil);
    resume (push_i ev)
```

```
let affinely_{n,i} = handler
  push_i ev resume ↦
   //replaces ev's lifetime in memory:
    set_i (update (get_i ⟨⟩) ev (fin 1));
    resume (push_i ev)
```

Fig. 12. Simple Restriction Handlers.

```
1 let suspendable_i = handler
2   push_i ev resume ↦
3     push_i ev;
4     match peek_i ⟨⟩
5       true ↦ resume ⟨⟩
6       false ↦ stash_i (async_thread (resume)); ⟨⟩
```

```
1 type Strand_i = Maybe[{Unit}^⟨push_i,stash_i,peek_i,async⟩]
2 let play_pause_i =
3   handler(run: Bool = true, strand: Strand_i = none)
4     pause_i ⟨⟩ resume ↦ resume ⟨⟩ false strand
5     play_i ⟨⟩ resume ↦ match ⟨run, strand⟩
6             ⟨false, some k⟩ ↦ k ⟨⟩; resume ⟨⟩ true none
7             ⟨false, none⟩ ↦ error ⟨⟩ //illegal state
8             _ ↦ resume ⟨⟩ run strand
9     stash_i k resume ↦ resume ⟨⟩ run (some k)
10    peek_i ⟨⟩ resume ↦ resume run run strand
```

Fig. 13. Handlers for suspending/resuming interleaved iterations.

Due to space limitations, we elide the definition of the effect handler *gen*. Its responsibility is transforming **yield**ed values from effects back to ordinary data values:

$$gen\colon \forall\mu.\{\mathsf{Unit}\}^{\langle \mathbf{yield},\mathbf{fail},\mathbf{async},\mu\rangle} \to^{\langle \mathbf{async},\mu\rangle} \mathsf{R}[T_{n+1}],$$

so that other correlations may query the results.

4.4 Implementing Restriction Handlers

Without restrictions, the $join\_shape_n$ (Figure 10) computes the n-way cartesian product, where time and space requirements grow arbitrarily large. One or more restriction handlers must be assigned to the implicit variable ?*restriction* (cf. Section 2.4), in order to obtain a different correlation computation. By default, ?*restriction* is bound at the top level to an identity **handler** $\{x \mapsto x\}$, which does not influence the computation. Now, we define concrete implementations of the restriction handlers from Section 2.4.

4.4.1 Simple Restrictions. The $mostRecently_i$ handler (Figure 12, left-hand side) simply truncates all but the last observed event from the ith memory state. That is, it effectively restricts the memory for the ith input reactive to one cell.

The $affinely_i$ handler (Figure 12, right-hand side) sets the lifespan counter of events to 1, i.e., each event value from the i-th input reactive can occur in at most one pattern match.

4.4.2 Advanced Restriction Handlers: Play/Pause Iterations. One of the design challenges we identified in Section 1 is controllable matching behavior. We enable it with generic abstractions that coordinate the interleaved iteration over reactives (Definition 4.1), e.g., for expressing the join behavior of *zip* (Section 2.4).

We implemented the capability to suspend/resume individual *eat* r_i ($\mathbf{push}_i$) iterations for join computations. For example, we can make the i-th iteration suspendable as follows:

$$\{\mathbf{with}\ suspendable_i\ (eat\ r_i\ (\mathbf{push}_i))\}.$$

The $suspendable_i$ handler (Figure 13, left-hand side) implements a simple interception of the $\mathbf{push}_i$ effect (Line 3), which originates from the underlying iteration. That way, we can capture the continuation of the iteration in *resume* and store it for later, if the surrounding correlation

```
1 type Sᵢ = Maybe[Ev[Tᵢ]]
2 let aligning⟨i,j⟩ = handler(sᵢ: Sᵢ = none, sⱼ: Sⱼ = none)
3  pushᵢ ev resume ↦
4    pauseᵢ ⟨⟩; tryRelease⟨i,j⟩ (some ev) sⱼ (resume)
5  pushⱼ ev resume ↦
6    pauseⱼ ⟨⟩; tryRelease⟨i,j⟩ sᵢ (some ev) (resume)
```

```
1 let tryRelease⟨i,j⟩ = λsᵢ.λsⱼ.λresume.
2  match ⟨sᵢ, sⱼ⟩
3    ⟨some ev₁, some ev₂⟩ ↦  //synchronization complete
4      resume ⟨⟩ none none;   //stashes current strand
5      pushᵢ ev₁; pushⱼ ev₂;   //sequentially process events
6      play₁ ⟨⟩; play₂ ⟨⟩      //resume iterations
7    _ ↦ resume ⟨⟩ sᵢ sⱼ      //synchronization incomplete
```

Fig. 14. Restriction handler for aligning interleaved iterations.

computation decides to suspend the iteration. Line 4 invokes the effect **peek**$_i$: Unit $\longrightarrow$ Bool, which signals whether the iteration continues or not. In the latter case (Line 6), *suspendable*$_i$ stores *resume* via the ambient state effect **stash**$_i$: $\{\text{Unit}\}^{\langle \textbf{push}_i, \textbf{stash}_i, \textbf{peek}_i, \textbf{async}\rangle} \longrightarrow$ Unit.[5] The function *async_thread* is part of the asynchrony implementation and ensures that the asynchronous thread context of *resume* is properly captured by **stash**$_i$.

The effects **peek**$_i$ and **stash**$_i$ are handled by the *play_pause*$_i$ handler (Figure 13, right-hand side), which carries the **stash**ed suspension state and the ith iteration's continuation in its parameters. Additionally, *play_pause*$_i$ handles the two commands **pause**$_i$: Unit $\longrightarrow$ Unit and **play**$_i$: Unit $\longrightarrow$ Unit. They provide an interface to the correlation computation for imperatively signalling the suspension/resumption of the ith iteration strand. **play**$_i$ resumes the stashed iteration, if previously **pause**$_i$ was invoked (Line 6). Importantly, other ongoing iterations in the interleaving remain unaffected and continue processing.

We can now implement a wide range of coordination strategies in the join computation, such as *aligning* from Section 2.4. The *aligning*$_{n,S}$ handler (Figure 14, left-hand side) causes an input subset $S \subseteq \{1, \ldots, n\}$, $|S| > 1$, to be iterated in lockstep. We exemplify it for two inputs i, j, which have the *suspendable* capability, but the code extends to more inputs in an obvious way. The handler implements a synchronization barrier on the interleaved iterations for inputs i and j. To this end, it implements a simple state machine: On each **push** notification from either input, we pause the underlying iteration and store the event in the handler parameter. Then, we invoke *tryRelease* (Figure 14, right-hand side) to check if the iterations are in lockstep. In this case, we forward the buffered event notifications further up the handler stack towards the cartesian implementation (Line 5). Once the surrounding handler context finished processing these two event values, Line 6 resumes the interleaved iteration.

To support suspension/resumption of all strands, minor modifications to the framework are necessary. One is replacing the ?*streams*$_n$ implicit binding in with iterations wrapped in *suspendable*. The other is applying the handlers $\boxplus_{i=1}^{n}$*play_pause*$_i$ immediately after *trigger*$_n$ in *join_shape*$_n$ (Figure 10).

4.4.3 Linear/Affine Effects. Handlers in λ_{cart} support by default multi-shot resumptions, which can have problematic interactions with resources. For instance, the **push**$_i$ effects need to be one-shot. Otherwise, future event notifications might be incorrectly duplicated. For these kinds of effects, a linear typing discipline would be appropriate [Wadler 1990b]. However, if a correlation incorporates **pause**$_i$/**play**$_i$, then **push**$_i$ requires an affine type. Integrating linear/affine types with algebraic effect handlers is an active area of research.

[5] Since *resume* accepts a unit value, it is a thunk of type $\{\text{Unit}\}^{\langle \textbf{push}_i, \textbf{stash}_i, \textbf{peek}_i, \textbf{async}\rangle}$.

```
let $slidingWindow_n = \lambda length.\lambda period.$
  handler
    $\mathbf{bind}_n\ \langle r_1, \ldots, r_n \rangle\ body \mapsto$
      let implicit $?pattern = body$ in
        with $(winArbiter_n\ length\ period\ \langle r_1, \ldots, r_n \rangle)$
            $((\{eat\ r_1\ (\mathbf{push}_1)\} \parallel \cdots \parallel \{eat\ r_n\ (\mathbf{push}_n)\})\ \langle\rangle)$
```

```
let $winArbiter_n = \lambda len.\lambda p.\lambda rs.$
  handler($win\_state$ = nil)
   $\{\mathbf{push}_i\ ev\ resume \mapsto$
     let $win\_state' = updateState\ len\ p\ win\_state\ rs\ ev$
     in $dispatch_i\ win\_state'\ ev;$
        $resume\ \langle\rangle\ win\_state'\ \}_{1 \le i \le n}$
```

Fig. 15. Handlers for Sliding Windows.

4.5 Windows

With effect handlers, windows (cf. Section 2.6) become a purely contextual restriction, which is orthogonal to join definitions, i.e., existing restriction handlers are reusable as-is. We give a brief sketch of sliding windows. Many other kinds of windows can be defined similarly.

We model a windowed correlation as a stateful computation that manages zero or more running copies of a $join\_shape_n$ instance (Figure 10), one per window, where windows may overlap. Figure 15, left-hand side defines the $slidingWindow_n$ delimiter. A declaration of a window overrides the default setup behavior of the $correlate_n$ form (Figure 11). The difference is that now handler $winArbiter_n$ directly interprets the $\mathbf{push}_i$ commands of the interleaved iteration.

$winArbiter_n$ (Figure 15, right-hand side) manages a set of active join shapes in its handler parameter win_state. Each time a new event value is pushed by the interleaved iteration, the *updateState* logic determines if new windows need to be allocated or expired ones need to be discarded, e.g., due to the passage of time.[6] Next, $dispatch_i$ multicasts the event to all active windows for which it is relevant. In the sliding window case, it is relevant if its occurrence time interval (Section 3.2) is entirely contained within a sliding window's start and end times. Iteration resumes once all relevant windows finished processing the event.

For each window instance, $winArbiter_n$ binds the interleaved iteration to a "facade", which is maintained by the *updateState* function. The facade consists of an interleaving of n streams, which are dynamically allocated per window.

5 VALIDATION OF CARTESIUS'S EXPRESSIVITY

This section validates our claim that Cartesius enables modeling and composing a range of join semantics across the domains of CEP/streaming engines, reactive programming languages/frameworks, as well as concurrent programming languages. We give a brief overview of different families of programming approaches for event correlation (Section 5.1) and survey a number of works across these families (Section 5.2) comparing their features with those of Cartesius.

5.1 Event Correlation Approaches

Complex Event Processing (CEP) features sequence patterns, aggregations and timing constraints for events [Demers et al. 2006; Diao et al. 2007]. In SASE+ [Agrawal et al. 2008], e.g., the following example pattern

```
PATTERN SEQ(Stock+ a[], Stock b)
WHERE skip_till_next_match(a[], b) {
  [symbol]
  and a[1].volume > 1000
  and a[i].price > avg(a[..i-1].price)
  and b.volume < 0.8*a[a.LEN].volume }
WITHIN 1 hour
```

[6]In this design, we take the time data from the event values to calculate time passage. Other designs are possible, such as active timer interrupts.

reports each stock that rises monotonically (Line 5, for all indices $i > 0$), *aggregating* the monotonic event sequence in `a[]`, where the rise is ended by an abrupt decline (Line 6). The scope of the pattern is delimited by a *sliding window* of 1 hour duration (Line 6). The `SEQ` combination of the `a[]` event sequence and the terminating `b` event is called a *complex event* (Line 1). All `a` and `b` events should refer to the same stock (Line 3).

Event correlations in CEP languages are often expressed in dedicated pattern languages on top of monolithic non-programmable runtime systems, i.e., the semantics are fixed and cannot be adapted to specific application (domain) needs. The semantics of CEP languages are ad-hoc variants of automata theory [Hopcroft et al. 2006], extended as needed to suit specific CEP systems.

Stream-relational algebra [Arasu et al. 2004, 2006; Arasu and Widom 2004] introduces streams as time-indexed relations and notions of time windows [Krämer and Seeger 2009]. Joins are specified in the relational algebra style. In CQL [Arasu et al. 2004], the following example

```
Select Istream(Close.item_id)
From Close[Now], Open[Range 5 Hours]
Where Close.item_id = Open.item_id
```

joins streams `Close` and `Open` representing end and start events for auctions, such that all auctions closed within 5 hours are reported. The distinction against CEP is not crisp and hybrid languages exist, i.e., embeddings of sequence patterns into CQL. However, such hybridization efforts are still an open research problem [Cugola and Margara 2012].

Reactive programming (RP) [Cooper and Krishnamurthi 2006; Czaplicki and Chong 2013; Elliott 2009; ReactiveX [n. d.]; Salvaneschi et al. 2014] exhibits crisp semantics and programming language embeddings. These languages feature some notion of first-class data flows, a.k.a. signals, which can be freely composed via arbitrary expressions. In FrTime [Cooper and Krishnamurthi 2006], the following example

```
(lift-strict (λ (y z) (/ y z))
             (posn-x mouse-pos) (width window))
```

joins the mouse's absolute x coordinate signal with the width signal of the GUI window into a signal computing the relative coordinate.

Changes of the values of the input signals cause the joint signal's value to be automatically re-computed. RP languages only exhibit specific join behaviors – typically, only the most recent values of the input signals are correlated – and it is not obvious how the richer join features of the CEP/stream domains translate into respective RP notions.

Concurrent programming languages feature synchronization patterns [Benton et al. 2004; Conchon and Le Fessant 1999; Fluet et al. 2008; Reppy 1991], where event joins are control structures for coordinating concurrent executions. The following JoCaml example [Mandel and Maranget 2014] defines join patterns [Fournet and Gonthier 1996] for discerning two interesting situations:

```
def wait() & finished(r) = reply Some r to wait
or  wait() & timeout()   = reply None to wait
```

Either a consumer's `wait` message coincides with a producer's `finished` message (top pattern) or the producer takes too long (bottom pattern), where a `timeout` occurs.

5.2 Survey

The survey is restricted to features supporting event joins, and is by no means exhaustive. Table 1 and Table 2 summarizes the surveyed works (rows) and feature categories (columns) related to joins. A checkmark (✓) indicates that a feature is readily available. Half-checked (∼) indicates that the feature is supported to a limited degree or can be implemented on top of the available abstractions. An empty ( ) cell indicates that a feature is not supported.

Table 1. Overview of supported join features in CARTESIUS and other works (1 of 2).

		Event Relations			Time Model		Windows					Event Selection					Event Consumption	
		Sequence	*Attribute*	*Timing*	*T. Stamp*	*Interval*	*Fixed*	*Sliding*	*Tumbling*	*Session*	*Custom*	*FIFO*	*LIFO*	*Position*	*Nondet.*	*Custom*	*Linear*	*Multiple*
CEP/Stream	Cayuga [Demers et al. 2006]	✓	✓	✓		✓						✓						✓
	Esper [Esper [n. d.]]	✓	✓	✓	✓		✓	✓	✓	✓	~	✓	✓	✓				✓
	EventJava [Eugster and Jayaram 2009]	✓	✓	✓	✓		✓		✓	✓		✓				~	✓	
	Rapide [Luckham et al. 1995]	✓	✓	✓	✓		✓	✓				✓						✓
	SASE+ [Agrawal et al. 2008; Diao et al. 2007]	✓	✓		✓			✓				✓			✓			✓
	TESLA [Cugola and Margara 2010]	✓	✓	✓	✓							✓	✓	✓			✓	✓
	Google Data Flow Model [Akidau et al. 2015]	✓	✓	✓	✓		✓	✓	✓	✓					✓	✓	✓	✓
	Snoop [Chakravarthy et al. 1994; Chakravarthy and Mishra 1994]	✓	✓		✓							✓			✓			✓
Async/RP	Asynchronous C♯ [Bierman et al. 2012]	✓	✓				~	~	~	~	~						✓	✓
	Elm [Czaplicki and Chong 2013]	✓	✓														✓	✓
	FrTime [Cooper and Krishnamurthi 2006] / Flapjax [Meyerovich et al. 2009]	✓	✓									✓	✓	✓	✓	✓	✓	✓
	Push/Pull FRP [Elliott 2009]	✓	✓	✓	✓	✓	~	~	~	~	~	~	~	~	~	~	✓	✓
	ReactiveX [ReactiveX [n. d.]]	✓	✓				~	~	~	~	~	~	~	~	~	~	✓	✓
Concur	CML [Reppy 1991]	~	✓	~											✓		✓	
	JoCaml [Conchon and Le Fessant 1999]		~												✓		✓	
	Polyphonic C♯ [Benton et al. 2004]														✓		✓	
	Manticore [Fluet et al. 2008]	~	✓	~											✓		✓	
	CARTESIUS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2. Overview of supported join features in CARTESIUS and other works (2 of 2).

		Ordering		Non-occurrence	Merge	Extensibility
		Temporal	*Custom*			
CEP/Stream	Cayuga	✓			✓	
	Esper	~	~	✓	✓	
	EventJava	✓				~
	Rapide				✓	
	SASE+	✓		✓	✓	
	TESLA	✓		✓		
	Google Data Flow Model	✓	✓		✓	
	Snoop	✓				
Async/RP	Asynchronous C♯	✓	✓			✓
	Elm	✓	✓			~
	FrTime / Flapjax	✓	✓		✓	✓
	Push/Pull FRP	✓	~	~	✓	✓
	ReactiveX	✓	✓	~	✓	✓
Concur	CML					✓
	JoCaml					✓
	Polyphonic C♯					✓
	Manticore					✓
	CARTESIUS	✓	✓		✓	✓

Event relations:

- *Sequence* – language abstractions specifying a notion of observation order between different streams of a join, e.g., “event a from s_1 is followed by event b from s_2”. Some languages specialize in detecting contiguous sequences, e.g., SASE+ [Agrawal et al. 2008], which we exemplified in Section 1. Other languages, e.g., TESLA [Cugola and Margara 2010], allow partial orders in a join, i.e., a subset of the input events must occur in a sequence, while others may occur in an arbitrary order.
- *Attribute* – constraints on the event contents, e.g., for filtering or associating by a common id. Most of the considered works readily support this feature. Exceptions are languages based on the Join Calculus [Fournet and Gonthier 1996], i.e., JoCaml [Mandel and Maranget 2014] and Polyphonic C♯ [Benton et al. 2004]. The former allows deep pattern matching on constructors in join patterns, but relating the contents of two or more events can only happen *after* the join triggers. This does not fit well together with the linear event consumption semantics (see below).
- *Timing* – the relative time distance between event pairs, e.g., “match if events a and b occur at most 10ms apart from each other”.

Time model is concerned with specifying any time information attached to events. We distinguish between one dimensional time stamps and two dimensional intervals. The latter enables finer-grained distinctions in combination with sequence relations [White et al. 2007]. In the CEP and stream domain, time models and timing are much more common than in asynchronous reactive or concurrent programming. The time model of CEP and stream systems is tied to the architecture of the system for ordering guarantees. Additionally, event times are not always explicitly exposed in the specification language, e.g., in SASE+ and Snoop. Even though they often provide time-related signals, most reactive programming languages do not attach timing information to events. This is why their time-related columns are empty ( ).
Windows is concerned with limiting and duplicating the extent of queries:

- *Fixed* creates one absolute window, e.g., "all events from May 1 to May 30".
- *Sliding* creates possibly overlapping windows of a given size and period, e.g. "a window of 1 month each week". The period increment may be fixed or event-based.
- *Tumbling* is a non-overlapping sliding window, e.g., "a window per month (each month)".
- *Session* is event-driven, e.g., "a window from when user logs in until the user logs out".
- *Custom.*

Windows in RP can be built as abstractions. This is why the related columns are half checked (∼).

Event selection:

- *FIFO* selects the events in received order.
- *LIFO* prioritizes the most-recently seen events.
- *Positional* enables selecting events by their position e.g., "the first n events of stream s_1 with every other event of stream s_2".
- *Nondeterministic* indicates that the selection of events may be random, i.e., multiple choices are possible.
- *Custom.*

Event consumption:

- *Linear* indicates that each event is only consumed exactly once; e.g., the zip combinator is linear.
- *Multiple* indicates that each event may be consumed many times; e.g., in a cartesian product, an event in one stream is consumed as many times as the size of the other stream.

Ordering:

- *Temporal* indicates that events can be ordered by time.
- *Custom.*

Non-occurrence stands for features that enable expressing that some event(s) should *not* occur. To guarantee non-occurrence, one can add timing constraints or time windows that induce a finite waiting time or rely on anchoring events, e.g., "match a followed by b, where no c occurred in between". Such negative reasoning is specially challenging in asynchronous systems, and we leave it for future work.

Merge/union is concerned with fusing events of multiple streams into one stream. Even without anticipating it in our design, we could emulate disjunction as in CML, Manticore, JoCaml.

Extensibility is concerned with the questions: Can new abstractions be built in addition or on top of the existing abstractions? This is possible in all reactive and concurrent programming works, since they are language-embedded. Practically all surveyed CEP/stream systems are closed systems. EventJava is half-checked, because it is language-embedded, but joins cannot be extended meaningfully.

Table 3. Impact of restriction handlers on a three-way cartesian product. n is the total number of random input events, distributed evenly among three input reactives.

3-way cartesian	**memory** ($\frac{\text{events}}{\text{100 iterations}}$)	**throughput** ($\frac{\text{events}}{\text{sec}}$)	**#tuples**
$n = 3 \cdot 370$ (sic)	610.5	23.06	50653000
mostRecently			
$n = 3 \cdot 3700000$	3	279632.76	11099998
affinely			
$n = 3 \cdot 3700000$	≈ 0	301311.34	3700000
zip			
$n = 3 \cdot 3700000$	≈ 0	372513.73	3700000

Discussion. Caveat emptor: At first glance, Table 1 and Table 2 may suggest that Cartesius is superior in comparison. However, this is because in this paper, we focus on event joins only, while other works have a broader or different scope. We don't expect that the full feature set can or should be anticipated. Our message is that we can cover a range of features around event joins from the literature delivering a unifying model, based on a small foundation with clear semantics and an extensible design.

6 VALIDATION OF THE COMPUTATIONAL INTERPRETATION

In this section, we present the results of a preliminary study we conducted to gain insights on and quantify the potential of control flow restrictions imposed by effect handlers on joins for reducing the search space.

Setup and Metrics. We generated streams of random integer-valued events and passed them to four variants of a 3-way join in our multicore OCaml implementation processing: (1) an unrestricted cartesian product, (2) one with *mostRecently* restriction on all inputs, (3) one with the *affinely* restriction on all inputs, and (4) a 3-way *zip* with both *align* and *mostRecently* restrictions (Section 4.4). We executed the benchmarks on a Mac Pro, 3 GHz Intel Xeon E5-1680 CPU, 32 GB system memory. For each join definition, we measured (1) the average number of events retained in the join's memory, sampled after every 100 **push** notifications, (2) the average throughput in events per second, and (3) the total number of candidate tuples generated.

Results. Table 3 summarizes the results. Note that for the pure cartesian product, we used $n = 3x370$ as opposed to $n = 3x370000$ for the other join variants. This reduction of the input size by several orders of magnitude was necessary due to the high computational cost of the pure cartesian product. While they process 10.000 times more events per each input, the variants of the 3-way joins with effect handlers perform significantly faster and with negligible memory overhead:
(1) The memory consumption reflects the specified join semantics, e.g., *mostRecently* retains exactly one event for each input and *affinely* on average retains no event (by contrast the cartesian product join retains 610 events for the inputs).
(2) Likewise, joins restricted by effect handlers have a significantly higher throughput (between 12.000 and 16.000 times).
(3) The significant reduction of candidate tuples generated by the *mostRecently*, *affinely*, and *zip* joins (between 5 and 13 times), demonstrates that effect handlers effectively avoid irrelevant computations, which leads to the significant memory reductions and throughput improvement.

In summary, these results demonstrate that our effect handlers impose adequate control flow restrictions to avoid needless materialization of tuples.

7 APPROXIMATING GENERATIVE EFFECTS

We have shown in Section 4 how to embed Cartesius into a λ-calculus with algebraic effects and handlers. There is a semantic gap to cross when deriving executable implementations from our design, in particular concerning generativity and typing (Section 4.2). In this section, we discuss how such conceptual challenges impacted the embedding of Cartesius into Koka [Leijen 2017b] and multicore OCaml [Dolan et al. 2017], two state of the art implementations of algebraic effects.

The definitions in Section 4 are generative in the sense that they should be bound to a join definition, which is a value. Two different join definitions should have their own, non-overlapping **push** effects, because they can have different arities and input types.

Both languages have rather complimentary strengths: Koka has strong effect typing, but has no support for generative effects. On the other hand, multicore OCaml has no effect typing, but generativity can be encoded by using OCaml's powerful module system.

Our multicore OCaml implementation, encodes generative effects with the built-in module system.[7] For example, a **push** effect is part of a module interface

```
module type SLOT = sig
  type t
  effect Push: t -> unit
end
```

and this is how we put it to use, by the example of a very simplified binary join definition, which is a functor:

```
module Join2(T: sig type t0 type t1 type result end) =
struct
  module I0 = Slot(struct type t = T.t0 evt end)
  module I1 = Slot(struct type t = T.t1 evt end)
  effect Trigger: S0.t * S1.t -> unit
  let trigger p = perform (Trigger p)
  let reify action =
     try action () with
     | effect (I0.Push v) k ->
        forEach trigger (cartesian0 v); gc (); continue k ()
     | effect (I1.Push v) k ->
        forEach trigger (cartesian1 v); gc (); continue k ()
   (* ... *)
end
```

Note that two different slot instances I0 and I1 are allocated, each describing its own **push** effect. The **trigger** effect's type of the join definition is dependent on the two slot instances type members. We can discern the two different push effects, because they are defined in two separate module instances.

In Koka, the push effect would need to be defined using a polymorphic effect interface:

```
effect cart<a> { push(index: int, v: a): () }
```

where the command carries an extra index parameter to discern which reactive emitted an event. Now, a handler clause has to compare the index argument at runtime to determine whether it is responsible for handling a push effect:

```
handler {
  push(i, v) ->
    if (equal(i,1)) then
      forEach trigger (cartesian1 v); gc (); resume ()
    else resume (push(i,v)) }
```

[7] We thank Matija Pretnar for pointing out the encoding in a personal correspondence.

Another drawback of the Koka implementation is that unlike the multicore OCaml version, it does not support multiple instantiations of the `cart<a>` interface with different types, e.g., to join reactives of type `Int` and `String`. Due to parametricity, handlers for `cart<a>` must treat the pushed event values uniformly. Given that Koka has no notion of type classes, this is quite limiting, e.g., when printing event values for debugging purposes. We were forced to give up polymorphism for the `cart<a>` effect interface, fixing the type of reactives to `Int`.

We conclude that generativity is useful, but not yet well supported in current languages with built-in algebraic effects. We hope that our uses cases inspire new language designs for generativity.

8 RELATED WORK

To the best of our knowledge, this is the first work to propose algebraic effects and handlers as a common substrate to model semantic variants of reactive computations. In this section, we compare our work against other approaches that are similar in scope, i.e., unifying models.

Brooklet by Soulé et al. [Soulé et al. 2010] is a calculus for stream processing languages that models stream computations as static, asynchronous and acyclic data flow graphs. For example, stream-relational algebra languages, such as CQL [Arasu et al. 2006] and batch processing languages can be compiled into the calculus. The calculus emphasizes the topology of stream computations, which makes it well-suited as an intermediate representation for domain-agnostic optimizations at the operator graph level [Hirzel et al. 2013]. The internal behavior of a graph node remains a "black box" in the form of a deterministic pure state transition function, acting on a (possibly shared) state and a number of input queues. Cartesius is similar in that there is a strict separation between impurity and purity, mediated by algebraic effects. However, Brooklet itself has no means of specifying the internal combination behavior of nodes and is not language embedded. Cartesius is complementary, enabling the declaration of the behavior inside the black boxes. Coordination and state sharing among nodes can be achieved by lexically enclosing with suitable handlers.

Streams *à la carte*, by Biboudis et al. [Biboudis et al. 2015], abstract over the semantics of stream pipelines using Oliveira et al.'s object algebra approach [Oliveira and Cook 2012; Oliveira et al. 2013], which models algebras in terms of OO interfaces and generics. Algebra instances are similar to handlers in the sense that both are modular folds over a given signature of operations. However, object algebras do not capture delimited control and must be passed as an explicit parameter to programs. The design by Biboudis et al. facilitates switching backend to execute a stream program, e.g., between push and pull streams. In contrast, Cartesius fixes the choice to Elliott's push/pull streams [Elliott 2009] to support direct style correlation patterns in consumers, even though event production is push-based. This works well in combination with effect handlers, because of their ability to capture the continuation.

Stream fusion, to completeness by Kiselyov et al. [Kiselyov et al. 2017], uses staging to remove abstraction overhead of a range of pull-based stream combinators. The generated primitive, loop-based code rivals hand-written implementations. Pull-based streams can be thought of as generators or unfolds. A key finding of Kiselyov et al. is staging of zip – which is complex, because the two streams need to be advanced in lock step, yet a (non-linear) stream may need to be advanced a statically unknown number of times to produce one element. Their solution involves ensuring that one of the streams is linear through reification; they argue that there is no way around this inefficiency, even with hand-written code. In our case, we already reify the memory states, so aligning them for zipping is not particularly difficult. Yet, a thorough assessment of the performance of our solution is left for future work.

The SECRET model [Botan et al. 2010; Dindar et al. 2013] examines windows in stream processing engines and provides a unifying description of vastly different windowing behaviors among varied systems. In Cartesius, we model windows as contextual restrictions (see Section 4.5), and leave the exact behavior of how events are dispatched to windows (e.g., sliding) open to implementors.

Ziarek et al. [2011] contribute a composable and extensible design for asynchronous events, based on the Concurrent ML API. Their work targets general purpose, asynchronous programming, while the focus of Cartesius is on joining asynchronous event streams. Nevertheless, some notable similarities exist to handler-based event correlation (Definition 4.1). Handling the interleaved iterations is similar to applying CML's choose combinator to CML events. The effect handlers for **push** correspond to the case analysis of choose. Among the differences are that choose is one shot, defining a single event, whereas Cartesius continuously applies the case analysis to event sequences, producing multiple events. Moreover, in our context, events are effect invocations and not the events arriving on the streams being composed. Finally, their work supports parallel executions, whereas Cartesius so far only supports single step, interleaving concurrency.

9 CONCLUSION

We used algebraic effects and handlers to reduce the problem of expressing diverse join features to the problem of writing modular effect handlers that influence a cartesian product evaluation. Operationally, event correlation can be understood in terms of specific selection and consumption patterns on a local memory of event observations. We exploited this insight to avoid needless materialization of event tuples, demonstrating that efficiency is achievable despite having a design based on generic, modular components.

A TYPE SYSTEM

Expression Typing

$$\boxed{\Gamma \vdash e\colon T \,|\, \varepsilon}$$

$$\Gamma, x{:}T, \Gamma' \vdash x\colon T \,|\, \langle\rangle \qquad \frac{\Gamma, x{:}T_1 \vdash e\colon T_2 \,|\, \varepsilon}{\Gamma \vdash \lambda x.e\colon T_1 \to^{\varepsilon} T_2 \,|\, \langle\rangle} \qquad \frac{\Gamma, \alpha^{\kappa} \vdash e\colon T \,|\, \langle\rangle}{\Gamma \vdash \Lambda\alpha^{\kappa}.e\colon \forall\alpha^{\kappa}.T \,|\, \langle\rangle} \qquad \frac{\mathsf{k}\ (T_i)_{1\le i\le n} \in D\ \overline{S} \qquad (\Gamma \vdash e_i\colon T_i \,|\, \varepsilon)_{1\le i\le n}}{\Gamma \vdash \mathsf{k}\ (e_i)_{1\le i\le n}\colon D\ \overline{S} \,|\, \varepsilon}$$

$$\frac{com_{\overline{S}}\colon T_1 \to T_2}{\Gamma \vdash com_{\overline{S}}\colon T_1 \to^{\langle com_{\overline{S}}\rangle} T_2 \,|\, \varepsilon} \qquad \frac{\Gamma \vdash e_1\colon T_1 \to^{\varepsilon} T_2 \,|\, \varepsilon \qquad \Gamma \vdash e_2\colon T_1 \,|\, \varepsilon}{\Gamma \vdash e_1\ e_2\colon T_2 \,|\, \varepsilon} \qquad \frac{\Gamma \vdash e\colon \forall\alpha^{\kappa}.T_1 \,|\, \varepsilon}{\Gamma \vdash e\ [T_2^{\kappa}]\colon T_1[T_2^{\kappa}/\alpha^{\kappa}] \,|\, \varepsilon} \qquad \frac{\Gamma \vdash e\colon T \to^{\varepsilon} T \,|\, \varepsilon}{\Gamma \vdash \mathbf{fix}\ e\colon T \,|\, \varepsilon}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e\colon T_1 \,|\, \varepsilon & (\Gamma \vdash p_i\colon T_1, \Gamma_i)_{1\le i\le n} \\ (\Gamma, \Gamma_i \vdash e_i\colon T_2 \,|\, \varepsilon)_{1\le i\le n} & (p_i)_{1\le i\le n}\ \text{covers}\ T_1\end{array}}{\Gamma \vdash \mathbf{match}\ e\ \{(p_i \mapsto e_i)_{1\le i\le n}\}\colon T_2 \,|\, \varepsilon} \qquad \frac{\Gamma \vdash e\colon T_1 \,|\, \langle \overline{com}, \varepsilon\rangle \qquad \Gamma \vdash h\colon T_1 \ {}^{\langle\overline{com}\rangle}\!\!\Rightarrow^{\varepsilon} T_2}{\Gamma \vdash \mathbf{handle}\ \{h\}\ e\colon T_2 \,|\, \varepsilon}$$

$$\frac{\Gamma \vdash e\colon T_1 \,|\, \varepsilon_1 \qquad \Gamma \vdash T_1 \le T_2 \qquad \Gamma \vdash \varepsilon_1 \le \varepsilon_2}{\Gamma \vdash e\colon T_2 \,|\, \varepsilon_2} \qquad \frac{\begin{array}{c}\Gamma, x_0{:}T_1 \vdash e_0\colon T_2 \,|\, \varepsilon \qquad (com_i\colon U_i \to V_i)_{1\le i\le n} \\ (\Gamma, x_i\colon U_i, r_i\colon V_i \to^{\varepsilon} T_2 \vdash e_i\colon T_2 \,|\, \varepsilon)\end{array}}{\Gamma \vdash \{x_0 \mapsto e_0; (com_i\ x_i\ r_i \mapsto e_i)_{1\le i\le n}\}\colon T_1 \ {}^{\langle com_1,\ldots,com_n\rangle}\!\!\Rightarrow^{\varepsilon} T_2}$$

Subtyping

$$\boxed{\Gamma \vdash T_1^{\kappa} \le T_2^{\kappa}}$$

$$\Gamma \vdash T^{\kappa} \le T^{\kappa} \qquad \Gamma \vdash \langle\rangle \le \varepsilon \qquad \Gamma \vdash \langle com_1, com_2, \varepsilon\rangle \le \langle com_2, com_1, \varepsilon\rangle \qquad \frac{\Gamma \vdash \varepsilon_1 \le \varepsilon_2}{\Gamma \vdash \langle com, \varepsilon_1\rangle \le \langle com, \varepsilon_2\rangle}$$

$$\frac{\Gamma \vdash T_1^{\kappa} \le T_2^{\kappa} \qquad \Gamma \vdash T_2^{\kappa} \le T_3^{\kappa}}{\Gamma \vdash T_1^{\kappa} \le T_3^{\kappa}} \qquad \frac{\Gamma, \alpha^{\kappa} \vdash T_1 \le T_2}{\Gamma \vdash \forall\alpha^{\kappa}.T_1 \le \forall\alpha^{\kappa}.T_2} \qquad \frac{\Gamma \vdash T_1' \le T_1 \qquad \Gamma \vdash \varepsilon_1 \le \varepsilon_2 \qquad \Gamma \vdash T_2 \le T_2'}{\Gamma \vdash (T_1 \to^{\varepsilon_1} T_2) \le (T_1' \to^{\varepsilon_2} T_2')}$$

ACKNOWLEDGMENTS

For feedback on this work, we thank Philipp Haller, Sven Keidel, Neel Krishnaswami, Daan Leijen, Sam Lindley, Matija Pretnar, KC Sivaramakrishnan and Philip Wadler.

This work is supported by the European Research Council (ERC, Advanced Grant No. 321217 and Consolidator Grant No. 617805) and by the German Research Foundation (DFG, SFB 1053 and SA 2918/2-1).

REFERENCES

Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient Pattern Matching over Event Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proceedings of the VLDB Endowment* 12 (2015), 1792–1803.

Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2004. CQL: A Language for Continuous Queries over Streams and Relations. In *Database Programming Languages.* Springer Berlin Heidelberg.

Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* (2006).

Arvind Arasu and Jennifer Widom. 2004. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record* (2004).

Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *Comput. Surveys* 45, 4 (2013).

Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

Nick Benton, Luca Cardelli, and Cédric Fournet. 2004. Modern Concurrency Abstractions for C#. *Transactions on Programming Languages and Systems (TOPLAS)* 26, 5 (2004), 769–804.

Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. 2015. Streams à la carte: Extensible Pipelines with Object Algebras. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP).*

Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' Play: Formalizing Asynchronous C♯. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, James Noble (Ed.).

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL* POPL (2018).

Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proceedings of the VLDB Endowment* (2010).

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *Proceedings of Symposium on Principles of Programming Languages (POPL) (POPL '14).* ACM, 361–372.

Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. 1994. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of International Conference on Very Large Data Bases (VLDB).*

S. Chakravarthy and D. Mishra. 1994. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering* 14, 1 (1994), 1 – 26.

James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *Proceedings of International Conference on Functional Programming (ICFP).*

Silvain Conchon and Fabrice Le Fessant. 1999. JoCaml: mobile agents for Objective-Caml. In *First and Third International Symposium on Agent Systems Applications, and Mobile Agents (ASAMA).*

Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP).*

Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the International Conference on Distributed Event-based Systems (DEBS).*

Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *Comput. Surveys* 44, 3 (2012), 15:1–15:62.

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI).*

Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009).

Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards Expressive Publish/Subscribe Systems. In *Proceedings of the 10th International Conference on Advances in Database Technology (EDBT'06).*

Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. 2007. *SASE+: An Agile Language for Kleene Closure Over Event Streams.* Technical Report. UMass Technical Report 07.

Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. 2013. Modeling the execution semantics of stream processing engines with SECRET. *The VLDB Journal* (2013).

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Proceedings of the Symposium on Trends in Functional Programming.*

Jonathan Edwards. 2009. Coherent Reaction. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell'09)*.

Esper. [n. d.]. Esper. http://www.espertech.com/esper/.

Patrick Eugster and K.R. Jayaram. 2009. EventJava: An Extension of Java for Event Correlation. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*.

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103, 2 (1992), 235–271.

Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. 2008. Implicitly-threaded parallelism in Manticore. In *Proceedings of International Conference on Functional Programming (ICFP)*.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.* 1, ICFP (2017).

Cédric Fournet and Georges Gonthier. 1996. The reflexive CHAM and the Join-calculus. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.

Philipp Haller and Heather Miller. 2013. RAY: Integrating Rx and Async for Direct-Style Reactive Streams. (2013).

Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. 2013. A catalog of stream processing optimizations. *Comput. Surveys* (2013).

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2006. *Automata theory, languages, and computation.* Pearson.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *Proceedings of International Conference on Functional Programming (ICFP)*.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of Haskell Symposium.*

Jürgen Krämer and Bernhard Seeger. 2009. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems* 34, 1 (2009), 1–49.

Neelakantan R. Krishnaswami. 2013. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of International Conference on Functional Programming (ICFP)*.

Daan Leijen. 2017a. Structured Asynchrony with Algebraic Effects. In *Proceedings of the International Workshop on Type-Driven Development (TyDe)*.

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.

Sam Lindley. 2014. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014.*

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.

Hai Liu and Paul Hudak. 2007. Plugging a Space Leak with an Arrow. (2007), 29 – 45. Festschrift honoring Gary Lindstrom on his retirement from the University of Utah after 30 years of service.

David C. Luckham. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc.

D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. 1995. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 21, 4 (1995), 336–354.

Louis Mandel and Luc Maranget. 2014. The JoCaml language, Documentation and user's manual, Chapter on Concurrent programming. http://jocaml.inria.fr/doc/concurrent.html.

Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Neil Mitchell. 2013. Leaking Space. *ACM Queue* 11, 9, Article 10 (Sept. 2013), 14 pages.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Bruno C.d.S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP).*

Bruno C.d.S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*.

Tomas Petricek, Alan Mycroft, and Don Syme. 2011. Extending monads with pattern matching. In *Proceedings of Haskell Symposium*.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A calculus of context-dependent computation. In *Proceedings of International Conference on Functional Programming (ICFP)*.

Tomas Petricek and Don Syme. 2011. Joinads: A Retargetable Control-Flow Construct for Reactive, Parallel and Concurrent Programming. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*.

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* (2003).

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*.

ReactiveX. [n. d.]. ReactiveX. http://reactivex.io.

John H. Reppy. 1991. CML: A Higher-Order Concurrent Language. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*.

Rx.NET Example. 2013. Event Correlation. https://github.com/dotnet/reactive/blob/master/Rx.NET/Samples/EventCorrelationSample/EventCorrelationSample/Program.cs#L55

Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the International Conference on Modularity*. ACM.

Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A Universal Calculus for Stream Processing Languages. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. Springer-Verlag.

Wouter Swierstra. 2008. Data types à la carte. *Functional Programming* (2008).

Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# Asynchronous Programming Model. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*.

Philip Wadler. 1990a. Comprehending monads. In *LISP and Functional Programming*.

Philip Wadler. 1990b. Linear types can change the world. In *Proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods*.

Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*.

Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. 2007. What is "Next" in Event Processing?. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*.

Lukasz Ziarek, K. C. Sivaramakrishnan, and Suresh Jagannathan. 2011. Composable asynchronous events. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*.